RICE UNIVERSITY

Accelerated Discontinuous Galerkin Solvers with the Chebyshev Iterative Method on the Graphics Processing Unit

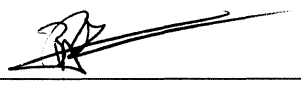
by

Toni Kathleen Tullius

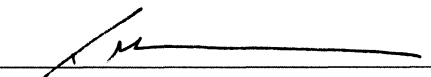
A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Arts

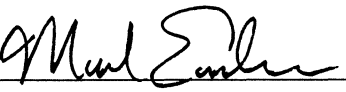
Approved, Thesis Committee:



Béatrice Rivière, Chair
Associate Professor of Computational and Applied Mathematics



Tim Warburton, Co-Chair
Associate Professor of Computational and Applied Mathematics



Mark Embree
Professor of Computational and Applied Mathematics

Houston, Texas

April, 2011

ABSTRACT

Accelerated Discontinuous Galerkin Solvers with the Chebyshev Iterative Method on the Graphics Processing Unit

by

Toni Kathleen Tullius

This work demonstrates implementations of the discontinuous Galerkin (DG) method on graphics processing units (GPU), which deliver improved computational time compared to the conventional central processing unit (CPU). The linear system developed when applying the DG method to an elliptic problem is solved using the GPU. The conjugate gradient (CG) method and the Chebyshev iterative method are the linear system solvers that are compared, to see which is more efficient when computing with the GPU's parallel architecture. When applying both methods, computational times decreased for large problems executed on the GPU compared to CPU; however, CG is the more efficient method compared to the Chebyshev iterative method. In addition, a constant-free upper bound for the DG spectrum applied to the elliptic problem is developed. Few previous works combine the DG method and the GPU. This thesis will provide useful guidelines for the numerical solution of elliptic problems using DG on the GPU.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my committee advisors Dr. Tim Warburton, Dr. Béatrice Rivière, and Dr. Mark Embree for their guidance and support throughout my work.

A special thanks to Dr. Richard Tapia, Alliances for Graduate Education and the Professoriate (AGEP), my CAAM peers, and NSF* for their continuous support throughout my graduate school career.

Also, thank you Theresa Chatman, AGEP coordinator, mentor, and friend; without you my experiences at Rice would not be the same and the AGEP program would not be as successful.

I want to thank my family for their unending encouragement and love, especially my mother, brother, and sister. I could not have done this work without any of you.

I dedicate this thesis to my angels: father, grandfather, grandmotherdear, and cousin, whose memories are constantly making me strive for my best.

*This work was supported by NSF Cooperative Agreement Number HRD-0450363.

Contents

Illustrations

Tables

Nomenclature

BLAS	Basic Linear Algebra Subprograms, a linear algebra library
CG	Conjugate gradient method
CPU	Central processing unit
CSR	Compressed sparse row format
CUBLAS	CUDA's version of BLAS
CUDA	Compute unified device architecture, a parallel computing language
CUSP	Sparse linear algebra and graph computations on CUDA
DG	Discontinuous Galerkin
FEM	Finite element method
FVM	Finite volume method
GPU	Graphics processing unit
LAPACK	Linear Algebra PACKage, a linear algebra library
NVIDIA	Corporation which specializes in the development of GPUs
PDE	Partial differential equations
λ	Eigenvalue: $\lambda \in \mathbb{R}$ such that $Ax = \lambda x$
spectrum	Space that contains all the eigenvalues
$H^s(\Omega)$	Sobolev space $H^s(\Omega) = \{v \in L^2(\Omega) : \forall 0 \leq \lvert\alpha\rvert \leq s, D^\alpha v \in L^2(\Omega)\}$
$v^T u$	Inner product: $v^T u = v_1 u_1 + v_2 u_2 + \cdots + v_n u_n$

$\|v\|_{L^2(\Omega)}$	Inner product space: $\|v\|_{L^2(\Omega)} = \left(\int_\Omega v^2\right)^{1/2}$
Ω	Polyhedral domain
$\partial\Omega$	Boundary of polyhedral domain
Γ_h	Set containing all interior and boundary edges of each element in T_h
T_h	Space that partitions the domain Ω into triangle elements
V_h	Discontinuous finite element space, $V_h = \{v \in L^2(\Omega) : v\|_E \in \mathbb{P}^N(E) \forall E \in T_h\}$
d	Dimension $d = 2$
N	Degree of the polynomial
$\mathbb{P}^N$	Space of polynomials of degree at most N
E	Element in mesh
∂E	Boundary of element
$\|\partial E\|$	Perimeter of an element
$\|E\|$	Area of an element
E_1^e, E_2^e	Two adjacent elements
e	Edge to an element
$\|e\|$	Length of an edge
h_E	Diameter of each element
h	h = $\max_{E \in T_h} h_E$
h_{sum}^E	$h_{sum}^E = \sum_{e \in \partial E} \frac{1}{h_E^i}$
M	Size of matrix
K	Symmetric matrix bounded for each element
σ_e	Penalty parameter over each edge, e
ϵ	Stability parameter, for SIPG $\epsilon = -1$
$[v]$	Jump: $[v] = v\|_{E_1^e} - v\|_{E_2^e}$
$\{v\}$	Average: $\{v\} = \frac{1}{2}\left(v\|_{E_1^e} + v\|_{E_2^e}\right)$

C_2	Coercivity constant
D_*	Constant such that $\|v\|_{DG} \leq D_* \|v\|_{L^2(e)}$
D_t	Constant given by trace inequalities, $D_t = \sqrt{\frac{(N+1)(N+2)}{2}}$

Chapter 1

Introduction

Supercomputers are popular because of their ability to allow data-intensive computations to perform in parallel, and therefore the computational time is drastically decreased. The discontinuous Galerkin (DG) method, a partial differential equation (PDE) solver, is becoming a preferred method because of its ability to allow the use of complex geometries. This work capitalizes on both features of DG and supercomputing by implementing a component of the DG method using a parallel computer. In addition to this implementation, computable bounds on the maximum and minimum eigenvalues associated with the DG scheme applied to the elliptic problem are analyzed, and a constant-free upper bound is developed. The lower bound remains an open problem.

When solving a PDE, there are many different approaches to choose from, ranging from the finite element method (FEM) to the finite volume method (FVM). Unlike the traditional FEM, the DG method allows piecewise discontinuous polynomials to represent the information of each element. Because of this, DG is useful when using unstructured meshes. Non-conforming meshes provide researchers with more flexibility when creating the discretized meshes and allow for better accuracy for their model.

The DG method was first proposed in the 1970s. Since then, mathematicians gradually found the benefits of the method, and more theory of the solver is known today. DG is already implemented using the languages C and MATLAB [1, 2]. Re-

search groups are now working on implementing DG using the graphics processing unit (GPU). The GPU is a parallel, multi-thread, many core processor that acts as a co-processor to the main central processing unit (CPU) [3]. Recently GPUs have attracted the community because of their 'peak compute capability and high memory bandwidth, in comparison to conventional CPUs' [4].

To take advantage of the potential of this new technology, this research implements a component of the DG method on the GPU, expecting to speed up computational time. This thesis solves the linear system, developed by DG, on the GPU. Because the GPU uses a parallel structure to execute commands, the use of inner products does not provide optimal performance. Therefore, the conjugate gradient (CG) method, a linear system solver with two inner products per iteration, may not be the most efficient solver. Another solver without inner products, the Chebyshev iterative method, will be tested and compared to CG. In order to use the Chebyshev iterative method, a quasi-tight approximation to the maximum and minimum eigenvalues of the discretized matrix is needed. A section of this research investigates bounds to the spectrum of the DG operator. This work establishes that the CG method, even with three inner products, is the optimal linear system solver when using the GPU architecture. This is because the CG method is guaranteed to converge with at most M iterations, where M is the size of the matrix. The Chebyshev iterative method can perform thousands of iterations before it converges.

A variety of applications ranging from flow and transport problems through a porous media to electromagnetics and wave propagation can be solved using the DG method. Mathematicians are currently working alongside engineers in the oil industry to show the many benefits that this method has. The work in this thesis applies the DG method to the elliptic problem. The Poisson equation has been seen

as one of the most prominent second order elliptic PDEs [5]. Fast Poisson solvers are needed to solve many practical equations for engineers, such as the heat conduction equation, the electrical field computation, and pressure correction in computational fluid dynamics [5]. With the completion of my research, the use of the DG method can become more appealing to engineers who specialize in any of the areas above, as my work focuses on reaping faster computational times.

The remainder of the thesis is as follows. Chapter 2 and Chapter 3 give more background on the DG method and GPU, respectively. Chapter 4 provides a brief overview of the iterative solvers used within this research. The DG model problem and analytical work for studying the bounds to the DG spectrum are presented in Chapter 5. Chapter 6 describes the method used to implement the iterative methods on the GPU. The numerical comparisons are presented in Chapter 7. Last, Chapter 8 gives concluding remarks about the research.

Chapter 2

Discontinuous Galerkin Method

The DG method is a numerical solver for finding a solution to a PDE. This method is said to be a combination of the FEM and FVM. More theory about the FEM and FVM solvers can be found in [6, 7]. Solving PDEs over an infinite domain can be difficult in obtaining an exact solution. Therefore, the domain is discretized to create a mesh of elements. Numerical PDE solvers approximate the solution of the PDE by creating polynomials representing information within each element. For most PDE solvers, like FEM, continuity of these polynomials between each element is required. However, for the DG method, no continuity restrictions across element boundaries are required, allowing for more complicated geometries and better accuracy. This research concentrates on two aspects associated with DG. First, this work implements a component of the DG method applied to the elliptic problem onto a supercomputer, causing a decrease in the computational time. Second, this thesis explores upper and lower bounds on the spectrum that is developed when applying the DG method.

The DG method was first developed in 1973 by Reed and Hill, in the framework of a neutron transport problem (determining the probability of a neutron-nuclear reaction occurrence) [8]. This scheme was designed mainly for hyperbolic equations. Around a similar time frame, discovered independently of Reed and Hill, discontinuous finite element methods were proposed for elliptic and parabolic equations [9]. Since then analysis for elliptic, parabolic, and hyperbolic equations has been extensively researched [10].

2.1 DG method versus other numerical solvers

Like the FEM and FVM, DG relies on creating a weak formulation of the equation, resulting in a simplified problem to solve and an approximation to closely match exact solution. For the DG method, there are few modifications compared to FEM's bilinear and linear forms of the variational problem to accommodate the discontinuities at the boundaries of the elements.

Two stabilizing terms are added to the bilinear form, a term corresponding to the fluxes and a penalty term [11]. There are different formulations of the numerical fluxes that have been developed over the years. Arnold, et al., conducted two studies that analyze the different numerical fluxes that have been introduced over the years [10, 12]. The choice of numerical fluxes will influence accuracy and stability of the method as well as properties of the stiffness matrix concerning sparsity and symmetry [10].

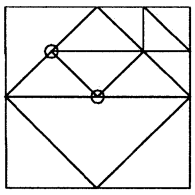


Figure 2.1 : DG can manage non-conforming meshes with hanging nodes (circled).

DG has several advantages compared to FEM and FVM. For one, because there are no continuity constraints between elements, DG methods are well suited to handle complicated geometries, i.e. non-conforming meshes with hanging nodes; see Figure 2.1. Also, DG can easily handle adaptivity strategies because refinement or unrefinement of the grid can be achieved without considering the continuity restrictions

typical for conforming in FEM [13]. DG methods are highly parallelizable [13]. Because of discontinuities of the elements, the mass matrix is block diagonal, easily invertible and simple to handle in parallel. The DG method also allows easy hp-analysis. In other words, within the DG scheme, one can have different polynomial degrees and different sizes for each element [1]. The concept of hp-analysis is easier to implement for DG versus FEM and FVM. Last, this method satisfies the local mass conservation, whereas FEM satisfies global mass balance over the whole domain [1]. All of these qualities are reasons why scientists are gaining interest in the DG method.

2.2 Pivotal results referenced in thesis

Recall that one aspect of this research is studying analysis for bounds on the eigenvalues associated with the DG scheme applied to the elliptic problem; refer to Chapter 5 for more details. Various works have been key to investigating approximations to the spectrum of the DG operator.

Antonietti proves the completeness and nonpollution of the spectrum [14], meaning that approximated maximum and minimum eigenvalues do exist. Another useful well known result is the Lax-Milgram theorem, which shows the existence and uniqueness of the DG variational problem as long as the bilinear form is continuous and coercive. The coercivity property explains that the bilinear form is bounded below by the DG-norm multiplied by a constant, C_2.

Epshteyn and Rivière [11], Shahbazi [15], independently, derive the coercivity constant, C_2, which is dependent on the polynomial degree and the angles of the mesh elements. More recently, Ainsworth and Rankin extended Shahbazi's penatly parameter by allowing complicated geometries for the mesh [16]. Within this paper, a

brief analysis for the lower bound estimate is given for the symmetric interior penalty Galerkin method. This thesis concentrates on trying to extend the bound created by the coercivity property, desiring a bound associated with the L^2-norm versus the DG-norm in order to find the approximation to the minimum eigenvalue. This bound still remains an open problem.

When exploring an upper bound approximation for the spectrum of the DG matrix, trace inequalities have proven essential. Traces describe information about the discontinuities at the boundary of each element in the mesh. Warburton and Hesthaven provide detailed analysis for deriving inverse trace inequalities for hp-finite elements [17]. In addition to those results, Ozisik, et al., present a tighter numerical bound to the Markov inequality [18]. All of these results were used in this research to find the approximation to the maximum eigenvalue to the DG operator.

The DG scheme applied to the elliptic problem has been implemented in both MATLAB and C [1, 2]. Full versions and details of the MATLAB code can be found online, developed by Warburton and Hesthaven [2]. This research will extend the implementation by using an interface called Compute Unified Device Architecture (CUDA), a parallel architecture that executes on the GPU. More information on CUDA and the GPU will be given in the next chapter.

Chapter 3

Graphics Processing Unit

The GPU is a 'highly parallel, multithread, manycore processor with tremendous computational horsepower and very high memory bandwidth' [3]. The evolution of the GPU was caused by the demand for real-time, high-definition 3D graphics. The GPU is appropriate when computations can be conducted in data-parallel processing, i.e. a function executes many commands simultaneously in parallel structure. Many programmers who deal with large data sets can use parallel computing in order to decrease computational time. One aspect to this research is to execute a component of the DG method applied to the elliptic equation on the GPU.

3.1 History of GPUs

Integrated circuits, graphics processing units, personal-computer motherboards, and video game consoles are all major products manufactured by NVIDIA, a multinational corporation from Santa Clara, California [19]. In the mid 1990s, there was an increase in public demand for hardware-accelerated 3D graphics, especially in the gaming industry [20]. Microsoft and Sony demand this new hardware for some of their video game consoles, including Xbox and Playstation 3. Initially, operating the hardware for the 3D graphics was not straightforward and only limited operations could be performed [20]. However, as technology progressed, more advances arrived. NVIDIA's GeForce256, originally released in 1999, is a graphics controller chip that has a GPU

[21]. The GeForce256's GPU can execute billions of calculations per second. This was a major advancement because developers had the chance to apply additional enhancements to features like character animation (physics) and advanced artificial intelligence (logic) [21]. This architecture was used until about 2006. The newest generation card is NVIDIA's Tesla C2050 [22]. This card is said to 'redefine high performance computing and make supercomputing available to everyone' [22].

In 2007, NVIDIA created software that allows users to operate the GPU as a co-processor to the CPU, where data-intensive, parallel tasks are executed simultaneously [23]. NVIDIA introduced a programming model called CUDA. This language allows communication between the CPU and the GPU.

3.2 Compute unified device architecture

The use of a GPU to perform general purpose engineering computations can be referred to as General Purpose GPU (GPGPU) [24]. NVIDIA revolutionized GPGPUs in 2006-2007 by introducing a new parallel language, CUDA. The concept of GPGPUs is to use the GPU in conjunction with the CPU to dramatically increase the performance compared to the conventional CPU, i.e the sequential part of the code runs on the CPU while the data-intensive part is executed in parallel by the GPU [24]. This architecture is a minimal extension of C and C++. The CUDA language is implemented in thousands of applications and published in research papers, including image and video processing, fluid dynamics simulations, CT image reconstruction, etc. [3].

Various PDE solvers are already executed using CUDA. In 2008, Zhao developed a lattice Boltzmann based algorithm that can be modified to solve elliptic Laplace and Poisson equations [25]. In 2010, Egloff explained how to implement finite difference

schemes for 1-D PDEs on the GPU [26]. There is also a team who is working on a software package, called FEAST [27]. This software is designed to solve PDEs using the FEM exploiting the floating point performance and memory bandwidth of the GPU. Few works have discussed linking DG to CUDA; therefore, this thesis helps provide useful guidelines. Klöckner, et al., are currently working on applying the CUDA architecture to solve Maxwell's equations on a general 3D unstructured grid using the DG method [28]. All works provide evidence that their implementations increase peak performance and decrease computation times. This thesis concentrates on working with the DG PDE solver associated with the elliptic problem.

Other useful papers that have guided this work are [4, 23, 29, 30], which describe optimal implementations for conjugate gradient. These papers also discuss using the Chebyshev iterative method as a pre-conditioner to the CG method [29, 30]. Li and Saad tested a sparse matrix-vector product kernel applied to pre-conditioned CG and GMRES methods [31]. Li and Saad's paper did not apply those linear solvers to DG. Also, Bell and Garland provide excerpts of code describing a sparse matrix-vector multiplication and provide ideas that are useful in implementing code for this research [32].

high performance sparse matrix-vector product (SpMV) kernels in different formats on current many-core platforms and used them to construct effective iterative linear solvers with several preconditioner options. Since the performance of triangular solve is low on GPUs, this computation can be accomplished by CPUs. By this hybrid CPU/GPU computations, IC preconditioned CG method and ILU preconditioned GMRES method are adapted to a GPU environment and achieve performance gains compared to its CPU counterpart

3.3 Characteristics of the GPU

One of the main reasons researchers want to code using CUDA is because of its advertisement in high peak performance; however, peak performance is not easily achieved. With this language, the programmer can control how the code is executed. To achieve high performance, careful consideration regarding the thread level parallelism and memory access methods while executing commands is necessary [4]. Another characteristic is that the peak performance associated with double precision is significantly less than that of single precision. For instance, the GPU card used within this research is NVIDIA's new Tesla C2050. The peak performance for single precision floating point operations is 1.03 $Tflops$, while double precision floating point performance is 515 $Gflops$ [22].

Another feature of the GPU is that CUDA has scientific libraries such as CUBLAS, which contains an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA driver [33]. This package is very user friendly and is used to perform many commands in this research.

The next trait of the GPU is a key element that is driving the research in this thesis. Recall that the GPU is used for running algorithms in parallel. However, the inner product is not optimal to solve in parallel. For example, the 2-norm of the vector, u, is calculated by:

$$u = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix},$$

$$\|u\|_2^2 = 1^2 + 2^2 + 3^2 + 4^2.$$

Each thread will compute the multiplication of each element but, when summing these values, the threads all have to communicate with each other in order to perform the

command. This forces the threads to synchronize, or to become serial in order to complete the inner product. Because of this trait, performing code with many inner products is not efficient.

When applying the DG method to the elliptic problem, a linear system needs to be solved. Solving the linear system is the most data-intensive part to the DG scheme, and this thesis concentrates on executing the linear system on the GPU. A comparison between two linear system solvers is provided, determining whether the use of an optimal solver with many inner products like the CG method is more efficient, or if using a different solver that optimizes the hardware and contains no inner products will be more favorable, i.e. the Chebyshev iterative method. The next chapter will give more details on these two iterative methods.

Chapter 4

Iterative Methods

When the DG scheme is applied to the elliptic problem, the linear system $Sx = b$ needs to be solved. One important task of this research is to implement the process of solving the linear system on the GPU, while decreasing the computational time when compared to solving on the CPU alone. Two different linear system solvers are tested: the CG method and the Chebyshev iterative method, i.e. a method with inner products versus a method without inner products.

4.1 The CG method

The CG method is a Krylov subspace method based on Lanczos algorithm. Consider the linear system, $Sx = b$. When S is a symmetric positive definite $M \times M$ matrix, the CG method can be applied. According to O'Leary, 'the conjugate gradient method is now the standard iterative method for solving linear systems involving sparse symmetric positive definite matrices' [34].

Before a definition of the CG method can be given, first define the standard form of the m-dimensional Krylov subspace where $m \leq M$ as

$$\mathcal{K}_m(S, r_0) = span\{r_0, Sr_0, \ldots, S^{m-1}r_0\},$$

where $r_0 = b - Sx_0$ is the residual for the initial guess, x_0. A Krylov subspace method computes iterates, x_k, of the form

$$x_k = x_0 + q_{k-1}(S)r_0,$$

where q_{k-1} is a polynomial of degree $k-1$.

The CG method can be described as a recurrence formula that generates unique iterates $x_k \in \mathcal{K}_k(S, r_0)$ and converges to $x_* = S^{-1}b$. At each step k, $\|e_k\|_S$ is minimized, where [35, 36]

$$e_k = x_* - x_k,$$
$$\|e_k\|_S = \sqrt{e_k^T S e_k}.$$

Define $\{p_0, p_1, \ldots, p_{k-1}\}$ as an S-conjugate basis, meaning $p_i^T S p_j = 0 \ \forall j \neq i$. Then [35]

$$\begin{aligned} \mathcal{K}_m(S, r_0) &= span\{r_0, Sr_0, \ldots, S^{m-1}r_0\} \\ &= span\{p_0, p_1, \ldots, p_{k-1}\} \\ &= span\{x_1, x_2, \ldots, x_k\} \\ &= span\{r_0, r_1, \ldots, r_{k-1}\}. \end{aligned}$$

where $x_k = x_{k-1} + \alpha_{k-1}p_{k-1}$ and α_{k-1} is some constant dependent on r_{k-1}, S, and p_{k-1}. More detailed information about the Krylov subspace and the CG method can be found in [35, 36, 37, 38].

There are many reasons for using CG. This method implicitly computes the best polynomial with respect to the eigenvalues of S [39]. Also, the CG method is guaranteed to converge with a maximum of M iterations. The rate of convergence of the CG method depends on the distribution of the eigenvalues of matrix, S. For polynomial degree k, the rate of convergence is approximately [36, 40]

$$2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k.$$

where $\kappa = \frac{\lambda_{max}}{\lambda_{min}}$.

However, the algorithm for this method involves the computation of two inner products and one matrix-vector product for every iteration (see Appendix A for the algorithm). As discussed above, implementing inner products in parallel requires the use of global communication, i.e thread synchronization; therefore, computing inner products with the CUDA language is not advised [29]. A discussion of the Chebyshev iterative method, a solver without inner products, is given in the next section.

4.2 The Chebyshev iterative method

The Chebyshev iterative method is used to solve the linear system $Sx = b$. This method, is a Krylov subspace method, creates a sequence of polynomials, $q_k(z)$, such that $|q_k([\alpha, \beta])| \leq 1$ and $q_k(0) = 1$ where $\alpha, \beta \in \mathbb{R}$ represents the minimum/maximum eigenvalue approximations to the matrix, S respectively. The polynomials used are the shifted and scaled Chebyshev polynomials. Recall the Chebyshev polynomials [35, 36, 37] is a three term recurrence relation where:

$$\begin{aligned} T_0(z) &= 1 \\ T_1(z) &= z \\ T_k(z) &= 2zT_{k-1}(z) - T_{k-2}(z) \;\; k = 2, 3, \ldots, \end{aligned}$$

where $|T_k(z)| \leq 1$ for $z \in [-1, 1]$. These polynomials oscillate in value between $[-1, 1]$. Using the Chebyshev polynomials on the interval $[\alpha, \beta]$, S^{-1} is written as:

$$S^{-1} = \frac{c_0}{2} + \sum_{k=1}^{k=\infty} c_k T_k(Z),$$

where $Z = \frac{2}{\beta-\alpha}[S - \frac{\beta+\alpha}{2}I]$, $c_k = \frac{1}{\sqrt{\alpha\beta}}(-\hat{q})^k$, and $\hat{q} = \frac{1-\sqrt{\alpha/\beta}}{1+\sqrt{\alpha/\beta}}$.

To find the solution to the linear system, this algorithm does not need knowledge about the initial guess used or the right hand side of the equation; however,

this method requires some knowledge about the spectrum. To use this algorithm, approximations to the maximum and minimum eigenvalues must be provided. If the approximation to maximum eigenvalue is an under-estimation, it is possible that the method may never converge. If the maximum eigenvalue approximation is an over-estimate, the method might take too long to converge. Tight approximations to the spectrum are necessary. The Chebyshev iterative method only works well for a well conditioned matrix, S. No spectrum that contains the origin can be used. The rate of convergence is determined by

$$\frac{2\hat{q}^k}{1+\hat{q}^{2k}}$$

where again $\hat{q} = \frac{1-\sqrt{\alpha/\beta}}{1+\sqrt{\alpha/\beta}}$.

Unlike the CG method, the Chebyshev iterative method avoids the use of inner products (refer to the algorithm in Appendix A). Because of this, when implemented on the GPU, the Chebyshev iterative method appears to be more efficient when parallelizing the code due to not having to calculate the inner products. However, this method may take more iterations to converge versus the CG method, which converges with a maximum of M iterations. This work determines whether using the Chebyshev iterative method is preferred compared to the CG method for the GPU.

The next chapter defines the model problem for the DG scheme when applied to the elliptic problem and investigates bounds for the maximum/minimum eigenvalues necessary for the Chebyshev iterative method.

Chapter 5

Finding Bounds on the DG Spectrum

The goal to this research is to compare two iterative linear system solvers, the conjugate gradient method and the Chebyshev iterative method, using the GPU, then combine these codes with a DG code that solves the elliptic problem. In order to perform the Chebyshev iterative method, approximations of the maximum/minimum eigenvalues of the DG operator are needed. The eigenvalue problem is to find $(0 \neq u, \lambda) \in V_h \times \mathbb{C}$ such that

$$a_h(u, v) = \lambda(u, v) \ \ \forall v \in V_h$$

(V_h is defined in the next section). The idea is to bound the spectrum and knowledge of the bounds exists due to of the result given by Antonietti, et al., [14] and the Lax-Milgram Theorem. Antonietti proves that the DG method is "spectrally correct" for a DG operator:

- non-pollution of the spectrum
- completeness of the spectrum
- non-pollution and completeness of the eigenspaces.

In other words, the paper [14] proves the properties above and in turn states that our spectrum is bounded. The bounds need to be precise in order to guarantee that the Chebyshev iterative method will converge with the optimal number of iterations. This chapter first defines the DG model problem for the general elliptic equation,

then analyzes the maximum/minimum bounds to the DG spectrum. The upper bound approximation to the spectrum is developed and the lower bound remains an open problem.

5.1 Model problem using DG method

5.1.1 Setup

Let $\Omega \in \mathbb{R}^2$ be the polyhedral domain, $\partial\Omega$ be the boundary of the domain. Denote $L^2(\Omega)$ as the Hilbert space with respect to the inner product and $H^s(\Omega)$ as the Sobolev space of order $s \geq 0$. The equations for all the definitions are below respectively, [1, 2, 14].

$$\begin{aligned} \|v\|_{L^2(\Omega)} &= \left(\int_\Omega v^2\right)^{1/2} \\ H^s(\Omega) &= \left\{v \in L^2(\Omega) : \forall 0 \leq |\alpha| \leq s, D^\alpha v \in L^2(\Omega)\right\} \end{aligned}$$

Let T_h be partitions of the domain, Ω, into triangles with possible hanging nodes. The parameter, h, is defined as

$$h = \max_{E \in T_h} h_E,$$

where h_E is the diameter of each element, $E \in T_h$. Let E_1 and E_2 be arbitrary elements in T_h; refer to Figure 5.1. The parameter ∂E_1 represents the boundary of element E_1. Denote Γ_I as the set of all interior edges and Γ_B as the set of all boundary elements. Then $\Gamma_h = \Gamma_I \cup \Gamma_B$. Also, define the discontinuous finite element space as V_h, given by

$$V_h = \{v \in L^2(\Omega) : v|_E \in \mathbb{P}^N(E)\ \forall E \in T_h\}$$

where $\mathbb{P}^N(E)$ is the space of polynomials of degree at most N on E.

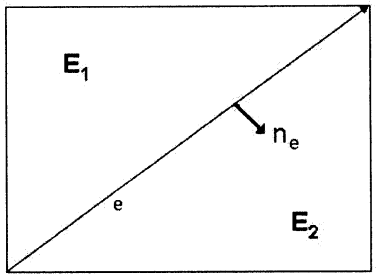


Figure 5.1 : Example of two adjacent triangular elements contained in a mesh

5.1.2 Jumps and averages

Recall from Chapter 2 that because DG allows discontinuities of the polynomials at the boundaries of the elements, this method incorporates ideas of numerical fluxes. Different derivations for numerical fluxes have been developed. For this research, jumps and averages are used to ensure the method will converge to the correct solution. Before defining jumps and averages, let us introduce the concept of trace. 'The notion of trace is used to define the restriction of a Sobolev function along the boundary of the element' [1].

Let $e \in \Gamma_I$, an interior face shared by E_1^e and E_2^e with n_e, the outward normal from E_1^e to E_2^e and let v be in V_h, refer to Figure 5.1. There are two traces along v for the two neighboring elements [1]. The jump and average for v can be defined respectively as:

$$\begin{aligned} [v] &= (v|_{E_1^e}) - (v|_{E_2^e}) \\ \{v\} &= \frac{(v|_{E_1^e} + v|_{E_2^e})}{2}. \end{aligned}$$

If $e \in \Gamma_B$ corresponding to element, E_1, then the jump and average can be written

as:

$$[v] = \{v\} = v|_{E_1^e}$$

5.1.3 Model problem

The general elliptic problem we consider is:

$$\begin{aligned} -\nabla \cdot (K\nabla u) + \alpha u &= f \text{ in } \Omega \\ u &= u_D \text{ on } \Gamma_D \\ K\nabla u \cdot n &= u_N \text{ on } \Gamma_N \end{aligned}$$

where K is a symmetric matrix that is bounded by constants, $k_1^E, k_2^E \in \mathbb{R}^+$ for every element, E,

$$k_1^E x^T x \leq x^T K x \leq k_2^E x^T x, \ \ \forall x \in E$$

$\alpha(x)$ is a non-negative scalar constant function, and $f \in L^2(\Omega)$. Also, u_D, u_N represent the Dirichlet and Neumann boundary conditions, respectfully, and $\Gamma_B = \Gamma_D \cup \Gamma_N$. Then the DG bilinear form, $a_h : V_h \times V_h \to \mathbb{R}$ is defined by

$$\begin{aligned} a_h(u,v) &= \sum_{E\in T_h} \int_E K\nabla u \cdot \nabla v + \int_\Omega \alpha uv - \sum_{e\in\Gamma_h} \int_e \{K\nabla u \cdot n_e\}[v] \\ &\quad +\epsilon \sum_{e\in\Gamma_h} \int_e \{K\nabla v \cdot n_e\}[u] + J_0^{\sigma_e,\beta_0}(u,v) \\ L(v) &= \int_\Omega fv + \epsilon \sum_{e\in\Gamma_D} \int_e (K\nabla v \cdot n_e + \frac{\sigma_e}{|e|^{\beta_0}} v) u_D + \sum_{e\in\Gamma_N} \int_e v u_N. \end{aligned}$$

The last term in $a_h(u,v)$, $J_0^{\sigma_e,\beta_0}(u,v)$, is a bilinear form that penalizes the jump of the function values:

$$J_0^{\sigma_e,\beta_0}(u,v) = \sum_{e\in\Gamma_h} \frac{\sigma_e}{|e|^{\beta_0}} \int_e [u][v].$$

The parameter σ_e is the penalty on each edge, e, that is nonnegative and real. The parameter, β_0, is a positive value that depends on the dimension, $d = 2$, $\beta_0(d-1) \geq 1$. For the rest of the research, $\beta_0 = 1$.

The parameter, ϵ, is the stability parameter. When $\epsilon = -1$, we obtain the Symmetric Interior Penalty method (SIPG). In this case, when the penalty, σ_e, is large enough, then the stiffness matrix is symmetric positive definite. For $\epsilon = 1$ we obtain the Non-Symmetric Penalty parameter method (NIPG). And for $\epsilon = 0$ we get the Incomplete Penalty parameter method (IPDG). For this research, the SIPG method is used. The next section will prove an upper bound to the variational problem.

5.2 Approximated upper bound of the spectrum

We have

$$\begin{aligned} a_h(v,v) = & \sum_{E\in T_h}\int_E K(\nabla v)^2 + \int_\Omega \alpha v^2 + (\epsilon-1)\sum_{e\in\Gamma_h}\int_e \{K\nabla v\cdot n_e\}[v] \\ & + \sum_{e\in\Gamma_h}\frac{\sigma_e}{|e|}\int_e [v]^2 \\ = & \sum_{E\in T_h} K\|\nabla v\|^2_{L^2(E)} + \alpha\|v\|^2_{L^2(\Omega)} + (\epsilon-1)\sum_{e\in\Gamma_h}\int_e \{K\nabla v\cdot n_e\}[v] \qquad (*) \\ & + \sum_{e\in\Gamma_h}\frac{\sigma_e}{|e|}\|[v]\|^2_{L^2(e)}. \end{aligned}$$

The next three subsections concentrate on further bounding the four terms on the right hand side by $\|v\|_{L^2(\Omega)}$.

5.2.1 Upper bound for the first term

This section bounds the first term in (*) such that

$$\sum_{E\in T_h} K\|\nabla v\|^2_{L^2(E)} \leq \hat{A}\|v\|^2_{L^2(\Omega)}.$$

where $\hat{A} \in \mathbb{R}$. In a paper written by Ozisik, et al., a tighter upper bound for Markov's inequality is presented [18]. The result of this paper is below.

Theorem 5.2.1. Markov Inequality for a planar triangle: *For a planar triangle, E, let $|\partial E|$ be the perimeter length of E and $|E|$ be the area of triangle E. Then, for a polynomial v of degree N,*

$$\|\nabla v\|_{L_2(E)} \leq \sqrt{C_N}\frac{|\partial E|}{|E|}\|v\|_{L^2(E)}$$

where the constants, C_N, are given in the paper [18]. Define h_E^i as the diameter corresponding to the i^{th} edge for $i = 1, 2, 3$. The parameter $\frac{|\partial E|}{|E|} = \sum_{e\in\partial E}\frac{2}{h_E^i}$, justification of parameter is given in Appendix B.

Using this result, then we have

$$K\|\nabla v\|^2_{L^2(E)} \leq KC_N\left(\frac{|\partial E|}{|E|}\right)^2\|v\|^2_{L^2(E)}.$$

Sum over all elements and take the maximum over the scalars, to obtain

$$\begin{aligned}\sum_{E\in T_h} K\|\nabla v\|^2_{L^2(E)} &\leq \sum_{E\in T_h} KC_N\left(\frac{|\partial E|}{|E|}\right)^2\|v\|^2_{L^2(E)} \\ &\leq C_N\max_{E\in T_h}\left(k_2^E\left(\frac{|\partial E|}{|E|}\right)^2\right)\sum_{E\in T_h}\|v\|^2_{L^2(E)} \\ &\leq C_N\max_{E\in T_h}\left(k_2^E\left(\sum_{e\in\partial E}\frac{2}{h_E^i}\right)^2\right)\sum_{E\in T_h}\|v\|^2_{L^2(E)}.\end{aligned}$$

Throughout the rest of the chapter use $h_{sum}^E = \sum_{e\in\partial E}\frac{1}{h_E^i}$, so that

$$\begin{aligned}\sum_{E\in T_h} K\|\nabla v\|^2_{L^2(E)} &\leq 4C_N\max_{E\in T_h}\left(k_2^E(h_{sum}^E)^2\right)\sum_{E\in T_h}\|v\|^2_{L^2(E)} \\ &\leq \hat{A}\sum_{E\in T_h}\|v\|^2_{L^2(E)},\end{aligned}$$

where $\hat{A} = 4C_N\max_{E\in T_h}\left(k_2^E(h_{sum}^E)^2\right)$.

5.2.2 Upper bound for third term

This section gives the bound for third term in (*) such that

$$(\epsilon - 1) \sum_{e \in \Gamma_h} \int_e \{K \nabla v \cdot n_e\}[v] \leq \hat{B} \|v\|^2_{L^2(\Omega)},$$

where $\hat{B} \in \mathbb{R}$. Then using the Cauchy-Schwarz inequality, we obtain

$$\sum_{e \in \Gamma_h} \int_e \{K \nabla v \cdot n_e\}[v] \leq \sum_{e \in \Gamma_h} \|\{K \nabla v \cdot n_e\}\|_{L^2(e)} \|[v]\|_{L^2(e)}. \qquad (**)$$

Warburton and Hesthaven proved two trace inequalities to bound this term, [17].

Theorem 5.2.2. Trace Inequality 1: *For some $\tilde{D}_t \in \mathbb{R}$ independent of h_E and v but dependent on the polynomial degree, N,*

$$\forall v \in \mathbb{P}_N(E),\ \forall e \subset \partial E,\ \ \|\nabla v \cdot n\|_{L^2(e)} \leq \tilde{D}_t \left(\frac{|e|}{|E|}\right)^{1/2} \|\nabla v\|_{L^2(E)}.$$

For $d = 2$, $\tilde{D}_t = \sqrt{\frac{N(N+1)}{2}}$, etc. Define h^i_E as the diameter associated with the i^{th} edge of the element, E, where $i = 1, 2, 3$. The parameter $\frac{|e|}{|E|} = \frac{2}{h^i_E}$; justification of parameter is given in Appendix B.

By applying the definition of average, the triangular inequality, Theorem 5.2.2, and by assuming K is scalar on each element then

$$\begin{aligned}
\|\{K \nabla v \cdot n_e\}\|_{L^2(e)} &\leq \frac{1}{2} \|K \nabla v \cdot n_e|_{E^e_1}\|_{L^2(e)} + \frac{1}{2} \|K \nabla v \cdot n_e|_{E^e_2}\|_{L^2(e)} \\
&\leq \frac{1}{2} k_2^{E^e_1} \|\nabla v \cdot n_e|_{E^e_1}\|_{L^2(e)} + \frac{1}{2} k_2^{E^e_2} \|\nabla v \cdot n_e|_{E^e_2}\|_{L^2(e)} \\
&\leq \frac{1}{2} k_2^{E^e_1} \tilde{D}_t \left(\frac{|e|}{|E^e_1|}\right)^{1/2} \|\nabla v\|_{L^2(E^e_1)} + \frac{1}{2} k_2^{E^e_2} \tilde{D}_t \left(\frac{|e|}{|E^e_2|}\right)^{1/2} \|\nabla v\|_{L^2(E^e_2)}.
\end{aligned}$$

Incorporating Theorem 5.2.1 from Ozisik et al. [18], leads to

$$\begin{aligned}\|\{K\nabla v\cdot n_e\}\|_{L^2(e)} \;\leq\; & \frac{1}{2}\tilde{D}_t\sqrt{C_N}k_2^{E_1^e}\left(\frac{|e|}{|E_1^e|}\right)^{1/2}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|_{L^2(E_1^e)}\\ & +\frac{1}{2}\tilde{D}_t\sqrt{C_N}k_2^{E_2^e}\left(\frac{|e|}{|E_2^e|}\right)^{1/2}\frac{|\partial E_2^e|}{|E_2^e|}\|v\|_{L^2(E_2^e)}.\end{aligned}$$

For the second term in (**), $\|[v]\|_{L^2(e)}$, a similar bound can be found using the second trace inequality of Warburton and Hesthaven [17].

Theorem 5.2.3. Trace Inequality 2: *For some $D_t \in \mathbb{R}$ independent of h_E and v but dependent on the polynomial degree, N,*

$$\forall v \in \mathbb{P}_N(E),\ \forall e \subset \partial E,\ \ \|v\|_{L^2(e)} \leq D_t\left(\frac{|e|}{|E|}\right)^{1/2}\|v\|_{L^2(E)}.$$

For $d = 2$, $D_t = \sqrt{\frac{(N+1)(N+2)}{2}}$,etc. Define h_E^i as the diameter corresponding to the i^{th} edge of element, E, where $i = 1, 2, 3$. The parameter $\frac{|e|}{|E|} = \frac{2}{h_E^i}$; justification of parameter is given in Appendix B.

Exploiting the the definition of jump, the triangle inequality and Theorem 5.2.3, we have

$$\begin{aligned}\|[v]\|_{L^2(e)} \;\leq\; & \|v|_{E_1^e}\|_{L^2(e)} + \|v|_{E_2^e}\|_{L^2(e)}\\ \leq\; & D_t\left(\frac{|e|}{|E_1^e|}\right)^{1/2}\|v\|_{L^2(E_1^e)} + D_t\left(\frac{|e|}{|E_2^e|}\right)^{1/2}\|v\|_{L^2(E_2^e)}.\end{aligned}$$

Applying the above bounds, the result becomes,

$$\begin{aligned}\int_e \{K\nabla v\cdot n_e\}[v] &\le \|\{K\nabla v\cdot n_e\}\|_{L^2(e)}\|[v]\|_{L^2(e)}\\ &\le \frac{1}{2}\tilde{D}_t D_t\sqrt{C_N}\left(k_2^{E_1^e}\left(\frac{|e|}{|E_1^e|}\right)^{1/2}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|_{L^2(E_1^e)}\right.\\ &\quad\left.+\,k_2^{E_2^e}\left(\frac{|e|}{|E_2^e|}\right)^{1/2}\frac{|\partial E_2^e|}{|E_2^e|}\|v\|_{L^2(E_2^e)}\right)\\ &\quad\times\left(\left(\frac{|e|}{|E_1^e|}\right)^{1/2}\|v\|_{L^2(E_1^e)}+\left(\frac{|e|}{|E_2^e|}\right)^{1/2}\|v\|_{L^2(E_2^e)}\right).\end{aligned}$$

Multiplying the terms out, we have,

$$\begin{aligned}\int_e \{K\nabla v\cdot n_e\}[v] &\le \frac{1}{2}\tilde{D}_t D_t\sqrt{C_N}\left(k_2^{E_1^e}\frac{|e|}{|E_1^e|}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|^2_{L^2(E_1^e)}\right.\\ &\quad+k_2^{E_2^e}\frac{|e|}{|E_2^e|}\frac{|\partial E_2^e|}{|E_2^e|}\|v\|^2_{L^2(E_2^e)}\\ &\quad+k_2^{E_1^e}\left(\frac{|e|}{|E_1^e|}\right)^{1/2}\left(\frac{|e|}{|E_2^e|}\right)^{1/2}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|_{L^2(E_1^e)}\|v\|_{L^2(E_2^e)}\\ &\quad\left.+\,k_2^{E_2^e}\left(\frac{|e|}{|E_1^e|}\right)^{1/2}\left(\frac{|e|}{|E_2^e|}\right)^{1/2}\frac{|\partial E_2^e|}{|E_2^e|}\|v\|_{L^2(E_1^e)}\|v\|_{L^2(E_2^e)}\right).\end{aligned}$$

Sum over the edges to obtain

$$\begin{aligned}\sum_{e\in\Gamma_h}\int_e \{K\nabla v\cdot n_e\}[v] &\le \sum_{E\in T_h}\sum_{e\in\partial E}\|\{K\nabla v\cdot n_e\}[v]\|_{L^2(e)}\\ &\le \frac{1}{2}\tilde{D}_t D_t\sqrt{C_N}\sum_{E\in T_h}\sum_{e\in\partial E}\left(k_2^{E_1^e}\frac{|e|}{|E_1^e|}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|^2_{L^2(E_1^e)}\right.\\ &\quad+k_2^{E_2^e}\frac{|e|}{|E_2^e|}\frac{|\partial E_2^e|}{|E_2^e|}\|v\|^2_{L^2(E_2^e)}\\ &\quad+k_2^{E_1^e}\left(\frac{|e|}{|E_1^e|}\right)^{1/2}\left(\frac{|e|}{|E_2^e|}\right)^{1/2}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|_{L^2(E_1^e)}\|v\|_{L^2(E_2^e)}\\ &\quad\left.+\,k_2^{E_2^e}\left(\frac{|e|}{|E_1^e|}\right)^{1/2}\left(\frac{|e|}{|E_2^e|}\right)^{1/2}\frac{|\partial E_2^e|}{|E_2^e|}\|v\|_{L^2(E_1^e)}\|v\|_{L^2(E_2^e)}\right)\end{aligned}$$

$$\begin{aligned}\sum_{e\in\Gamma_h}\int_e\{K\nabla v\cdot n_e\}[v] \;\leq\; & \frac{1}{2}\tilde{D}_t D_t\sqrt{C_N}\left(\sum_{E\in T_h}\sum_{e\in\partial E} k_2^{E_1^e}\frac{|e|}{|E_1^e|}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|^2_{L^2(E_1^e)}\right.\\ &+\sum_{E\in T_h}\sum_{e\in\partial E} k_2^{E_2^e}\frac{|e|}{|E_2^e|}\frac{|\partial E_2^e|}{|E_2^e|}\|v\|^2_{L^2(E_2^e)}\\ &+\sum_{E\in T_h}\sum_{e\in\partial E} k_2^{E_1^e}\left(\frac{|e|}{|E_1^e|}\right)^{1/2}\left(\frac{|e|}{|E_2^e|}\right)^{1/2}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|_{L^2(E_1^e)}\|v\|_{L^2(E_2^e)}\\ &\left.+\sum_{E\in T_h}\sum_{e\in\partial E} k_2^{E_2^e}\left(\frac{|e|}{|E_1^e|}\right)^{1/2}\left(\frac{|e|}{|E_2^e|}\right)^{1/2}\frac{|\partial E_2^e|}{|E_2^e|}\|v\|_{L^2(E_1^e)}\|v\|_{L^2(E_2^e)}\right).\end{aligned}$$

We developed bounds for the four terms on the right hand side separately. First, we bound the squared terms, starting with the term corresponding to E_1^e:

$$\begin{aligned}\sum_{E\in T_h}\sum_{e\in\partial E} k_2^{E_1^e}\frac{|e|}{|E_1^e|}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|^2_{L^2(E_1^e)} \;&\leq\; \sum_{E\in T_h}\sum_{e\in\partial E} k_2^{E_1^e}\max_{1\leq i\leq 3}\frac{2}{h^i_{E_1}}(2h^{E_1}_{sum})\|v\|^2_{L^2(E_1^e)}\\ &\leq\; 12\max_{E\in T_h}k_2^E\sum_{E\in T_h}\left(h^{E_1}_{sum}\max_{1\leq i\leq 3}\frac{1}{h^i_{E_1}}\right)\|v\|^2_{L^2(E_1^e)}.\end{aligned}$$

Take the maximum over all elements, then

$$\begin{aligned}\sum_{E\in T_h}\sum_{e\in\partial E} k_2^{E_1^e}\frac{|e|}{|E_1^e|}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|^2_{L^2(E_1^e)} \;&\leq\; 12\max_{E\in T_h}k_2^E\max_{E\in T_h}\left(h^E_{sum}\max_{1\leq i\leq 3}\frac{1}{h^i_E}\right)\sum_{E\in T_h}\|v\|^2_{L^2(E)}\\ &\leq\; \Theta\sum_{E\in T_h}\|v\|^2_{L^2(E)},\end{aligned}$$

where $\Theta = 12\max_{E\in T_h}k_2^E\max_{E\in T_h}\left(h^E_{sum}\max_{1\leq i\leq 3}\frac{1}{h^i_E}\right)$. Similarly for the second term,

$$\sum_{E\in T_h}\sum_{e\in\partial E} k_2^{E_2^e}\frac{|e|}{|E_2^e|}\frac{|\partial E_2^e|}{|E_2^e|}\|v\|^2_{L^2(E_2^e)} \;\leq\; \Theta\sum_{E\in T_h}\|v\|^2_{L^2(E)}.$$

Next, we bound the crossed terms. Define Ψ as:

$$\Psi = \sum_{E\in T_h}\sum_{e\in\partial E} k_2^{E_1}\left(\frac{|e|}{|E_1|}\right)^{1/2}\left(\frac{|e|}{|E_2|}\right)^{1/2}\frac{|\partial E_1|}{|E_1|}\|v\|_{L^2(E_1)}\|v\|_{L^2(E_2)}$$

Then

$$\begin{aligned}
\Psi &\leq \max_{E\in T_h} k_2^E \left(\sum_{E\in T_h}\sum_{e\in\partial E} \frac{|e|}{|E_1^e|}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|^2_{L^2(E_1^e)} \right)^{1/2} \left(\sum_{E\in T_h}\sum_{e\in\partial E} \frac{|e|}{|E_2^e|}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|^2_{L^2(E_2^e)} \right)^{1/2} \\
&\leq \max_{E\in T_h} k_2^E \left(12 \max_{E\in T_h} \left(h_{sum}^E \max_{1\leq i\leq 3} \frac{1}{h_E^i} \right) \sum_{E\in T_h} \|v\|^2_{L^2(E)} \right)^{1/2} \\
&\quad \times \left(\sum_{E\in T_h}\sum_{e\in\partial E} \frac{|e|}{|E_2^e|}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|^2_{L^2(E_2^e)} \right)^{1/2}.
\end{aligned}$$

Define $h_{sum} = \max_{E\in T_h} h_{sum}^E$. Then

$$\begin{aligned}
\sum_{E\in T_h}\sum_{e\in\partial E} \frac{|e|}{|E_2^e|}\frac{|\partial E_1^e|}{|E_1^e|}\|v\|^2_{L^2(E_2^e)} &\leq 2h_{sum} \sum_{E\in T_h}\sum_{e\in\partial E} \frac{|e|}{|E_2^e|}\|v\|^2_{L^2(E_2^e)} \\
&\leq 4h_{sum} \sum_{E\in T_h}\sum_{e\in\partial E} \max_{1\leq i\leq 3} \frac{1}{h_{E_2}^i}\|v\|^2_{L^2(E_2^e)} \\
&\leq 12h_{sum} \sum_{E\in T_h} \max_{1\leq i\leq 3} \frac{1}{h_{E_2}^i}\|v\|^2_{L^2(E_2^e)} \\
&\leq 12h_{sum} \max_{E\in T_h} \left(\max_{1\leq i\leq 3} \frac{1}{h_E^i} \right) \sum_{E\in T_h} \|v\|^2_{L^2(E)}.
\end{aligned}$$

So the bound for Ψ becomes

$$\begin{aligned}
\Psi &\leq \max_{E\in T_h} k_2^E \left(12 \max_{E\in T_h} \left(h_{sum}^E \max_{1\leq i\leq 3} \frac{1}{h_E^i} \right) \sum_{E\in T_h} \|v\|^2_{L^2(E)} \right)^{1/2} \\
&\quad \times \left(12h_{sum} \max_{E\in T_h} \left(\max_{1\leq i\leq 3} \frac{1}{h_E^i} \right) \sum_{E\in T_h} \|v\|^2_{L^2(E)} \right)^{1/2} \\
&\leq 12 \max_{E\in T_h} k_2^E \left(\max_{E\in T_h} \left(h_{sum}^E \max_{1\leq i\leq 3} \frac{1}{h_E^i} \right) \right)^{1/2} \left(h_{sum} \max_{E\in T_h} \left(\max_{1\leq i\leq 3} \frac{1}{h_E^i} \right) \right)^{1/2} \sum_{E\in T_h} \|v\|^2_{L^2(E)}
\end{aligned}$$

The last term is bounded identically. Putting all the bounds together, we have

$$\begin{aligned}(\epsilon-1)\sum_{e\in\Gamma_h}\int_e\{K\nabla v\cdot n_e\}[v] &\leq |\epsilon-1|\frac{1}{2}\tilde{D}_t D_t\sqrt{C_N}\left(24\max_{E\in T_h}k_2^E\max_{E\in T_h}\left(h_{sum}^E\max_{1\leq i\leq 3}\frac{2}{h_E^i}\right)\right.\\ &\quad +24\max_{E\in T_h}k_2^E\left(\max_{E\in T_h}\left(h_{sum}^E\max_{1\leq i\leq 3}\frac{1}{h_E^i}\right)\right)^{1/2}\\ &\quad \left.\times\left(h_{sum}\max_{E\in T_h}\left(\max_{1\leq i\leq 3}\frac{1}{h_E^i}\right)\right)^{1/2}\right)\|v\|^2_{L^2(\Omega)}\\ &\leq \hat{B}\|v\|^2_{L^2(\Omega)}.\end{aligned}$$

where $\hat{B}=12(\epsilon-1)\tilde{D}_tD_t\sqrt{C_N}\max_{E\in T_h}k_2^E\left(\max_{E\in T_h}\left(h_{sum}^E\max_{1\leq i\leq 3}\frac{1}{h_E^i}\right)\right.$
$\left.+\left(\max_{E\in T_h}\left(h_{sum}^E\max_{1\leq i\leq 3}\frac{1}{h_E^i}\right)\right)^{1/2}\left(h_{sum}\max_{E\in T_h}\left(\max_{1\leq i\leq 3}\frac{1}{h_E^i}\right)\right)^{1/2}\right)$.

5.2.3 Upper bound for the fourth term

This section bounds the last term in (*) such that

$$\sum_{e\in\Gamma_h}\frac{\sigma_e}{|e|}\|[v]\|^2_{L^2(e)}\leq\hat{C}\|v\|^2_{L^2(\Omega)}.$$

Using Warburton and Hesthaven's trace inequality, Theorem 5.2.3:

$$\begin{aligned}\|[v]\|_{L^2(e)} &\leq \|v|_{E_1^e}\|_{L^2(e)}+\|v|_{E_2^e}\|_{L^2(e)}\\ &\leq D_t\left(\frac{|e|}{|E_1^e|}\right)^{1/2}\|v\|_{L^2(E_1^e)}+D_t\left(\frac{|e|}{|E_2^e|}\right)^{1/2}\|v\|_{L^2(E_2^e)}.\end{aligned}$$

Square both sides of the inequality, then

$$\begin{aligned}\|[v]\|^2_{L^2(e)} &\leq D_t^2\frac{|e|}{|E_1^e|}\|v\|^2_{L^2(E_1^e)}+D_t^2\frac{|e|}{|E_2^e|}\|v\|_{L^2(E_2^e)}\\ &\quad +2D_t^2\left(\frac{|e|}{|E_1^e|}\frac{|e|}{|E_2^e|}\right)^{1/2}\|v\|^2_{L^2(E_1^e)}\|v\|^2_{L^2(E_2^e)}.\end{aligned}$$

Using the property $2ab\leq a^2+b^2$ for arbitrary a,b, then

$$\|[v]\|^2_{L^2(e)} \leq 2D_t^2\frac{|e|}{|E_1^e|}\|v\|^2_{L^2(E_1^e)}+2D_t^2\frac{|e|}{|E_2^e|}\|v\|^2_{L^2(E_2^e)}.$$

Multiply by the constant, $\frac{\sigma_e}{|e|}$, and sum over all edges, e, to obtain

$$\begin{aligned}
\sum_{e\in\Gamma_h}\frac{\sigma_e}{|e|}\|[v]\|^2_{L^2(e)} &\leq \sum_{E\in T_h}\sum_{e\in\partial E}\frac{\sigma_e}{|e|}\|[v]\|^2_{L^2(e)} \\
&\leq \sum_{E\in T_h}\sum_{e\in\partial E}\frac{\sigma_e}{|e|}\left(2D_t^2\frac{|e|}{|E_1^e|}\|v\|^2_{L^2(E_1^e)}+2D_t^2\frac{|e|}{|E_2^e|}\|v\|^2_{L^2(E_2^e)}\right) \\
&\leq 2D_t^2\left(\sum_{E\in T_h}\sum_{e\in\partial E}\frac{\sigma_e}{|e|}\frac{|e|}{|E_1^e|}\|v\|^2_{L^2(E_1^e)}+\sum_{E\in T_h}\sum_{e\in\partial E}\frac{\sigma_e}{|e|}\frac{|e|}{|E_2^e|}\|v\|^2_{L^2(E_2^e)}\right).
\end{aligned}$$

Notice for the term corresponding to E_1^e that

$$\begin{aligned}
\sum_{E\in T_h}\sum_{e\in\partial E}\frac{\sigma_e}{|e|}\frac{|e|}{|E_1^e|}\|v\|^2_{L^2(E_1^e)} &= \sum_{E\in T_h}\sum_{e\in\partial E}\frac{\sigma_e}{|E_1^e|}\|v\|^2_{L^2(E_1^e)} \\
&\leq \max_{e\in\Gamma_h}\sigma_e\sum_{E\in T_h}\sum_{e\in\partial E}\frac{1}{|E_1^e|}\|v\|^2_{L^2(E_1^e)} \\
&\leq 3\max_{e\in\Gamma_h}\sigma_e\sum_{E\in T_h}\frac{1}{|E_1|}\|v\|^2_{L^2(E_2^e)} \\
&\leq 3\left(\max_{e\in\Gamma_h}\sigma_e\right)\left(\max_{E\in T_h}\frac{1}{|E|}\right)\sum_{E\in T_h}\|v\|^2_{L^2(E)}.
\end{aligned}$$

For the term corresponding to E_2^e and identical bound is found. Then

$$\begin{aligned}
\sum_{e\in\Gamma_h}\frac{\sigma_e}{|e|}\|[v]\|^2_{L^2(e)} &\leq 12D_t^2\left(\max_{e\in\Gamma_h}\sigma_e\right)\left(\max_{E\in T_h}\frac{1}{|E|}\right)\sum_{E\in T_h}\|v\|^2_{L^2(E)} \\
&\leq \hat{C}\sum_{E\in T_h}\|v\|^2_{L^2(E)},
\end{aligned}$$

where $\hat{C}=12D_t^2\left(\max_{e\in\Gamma_h}\sigma_e\right)\left(\max_{E\in T_h}\frac{1}{|E|}\right)$.

5.2.4 Combining bounds

By using the values found in the previous sections, the upper bound to the variational problem becomes:

$$\begin{aligned} a_h(v,v) &= \sum_{E\in T_h} K\|\nabla v\|^2_{L^2(E)} + \alpha\|v\|^2_{L^2(\Omega)} + (\epsilon-1)\sum_{e\in\Gamma_h}\int_e \{K\nabla v\cdot n_e\}[v] \\ &\quad + \sum_{e\in\Gamma_h}\frac{\sigma_e}{|e|}\|[v]\|^2_{L^2(e)} \\ &\leq \hat{A}\sum_{E\in T_h}\|v\|^2_{L^2(E)} + \alpha\|v\|^2_{L^2(\Omega)} + \hat{B}\sum_{E\in T_h}\|v\|^2_{L^2(E)} + \hat{C}\sum_{E\in T_h}\|v\|^2_{L^2(E)} \\ &\leq (\hat{A}+\alpha+\hat{B}+\hat{C})\|v\|^2_{L^2(\Omega)}. \end{aligned}$$

Substituting the results, the upper bound becomes

$$\begin{aligned} a_h(v,v) \leq &\left(4C_N \max_{E\in T_h}\left(k_2^E (h_{sum}^E)^2\right) + \alpha\right. \\ &+12|\epsilon-1|\tilde{D}_t D_t\sqrt{C_N}\max_{E\in T_h} k_2^E\left(\max_{E\in T_h}\left(h_{sum}^E \max_{1\leq i\leq 3}\frac{1}{h_E^i}\right)\right. \\ &+\left.\left(\max_{E\in T_h}\left(h_{sum}^E\max_{1\leq i\leq 3}\frac{1}{h_E^i}\right)\right)^{1/2}\left(h_{sum}\max_{E\in T_h}\left(\max_{1\leq i\leq 3}\frac{1}{h_E^i}\right)\right)^{1/2}\right) \\ &\left.+\,12D_t^2\left(\max_{e\in\Gamma_h}\sigma_e\right)\left(\max_{E\in T_h}\frac{1}{|E|}\right)\right)\|v\|^2_{L^2(\Omega)}. \end{aligned}$$

5.2.5 Testing the bound

To check this bound, the Poisson solver for curved elements is used. Information about the implementation of this code can be found in [2]. Recall the Poisson equation

$$-\Delta u = f \; in \; \Omega$$

$$u = u_D \; on \; \Gamma_D$$

$$\frac{\partial u}{\partial n} = u_N \; on \; \Gamma_N$$

There is no α function, or matrix K, and it solves the SIPG method, therefore, $\epsilon = -1$. The upper bound becomes

$$\begin{aligned} a_h(v,v) &= \sum_{E\in T_h} \|\nabla v\|^2_{L^2(E)} + |2| \sum_{e\in\Gamma_h} \|\{\nabla v\cdot \mathbf{n}_e\}[v]\|_{L^2(e)} + \sum_{e\in\Gamma_h} \frac{\sigma_e}{|e|}\|[v]\|^2_{L^2(e)} \\ &\leq (\hat{A}+\hat{B}+\hat{C})\|v\|^2_{L^2(\Omega)} \\ &\leq \left(4C_N \max_{E\in T_h}(h^E_{sum})^2 + 24D_t^2\sqrt{C_N}\left(\max_{E\in T_h}\left(h^E_{sum}\max_{1\leq i\leq 3}\frac{1}{h^i_E}\right)\right.\right. \\ &\quad + \left(\max_{E\in T_h}\left(h^E_{sum}\max_{1\leq i\leq 3}\frac{1}{h^i_E}\right)\right)^{1/2}\left.\left(h_{sum}\max_{E\in T_h}\left(\max_{1\leq i\leq 3}\frac{1}{h^i_E}\right)\right)^{1/2}\right) \\ &\quad + 12D_t^2\left(\max_{e\in\Gamma_h}\sigma_e\right)\left(\max_{E\in T_h}\frac{1}{|E|}\right)\Bigg)\|v\|^2_{L^2(\Omega)}. \end{aligned}$$

Table 5.1 : Results for finding approximations to the maximum eigenvalue

N/h	0.2	0.12	0.1	0.05	0.025
1	1.1480	1.0709	1.0433	1.0714	1.0907
2	1.2409	1.1740	1.1446	1.1893	1.2122
3	1.2235	1.1853	1.1549	1.2019	1.2229
4	1.1992	1.1668	1.1378	1.1840	1.2103
5	1.1787	1.1521	1.1221	1.1676	1.1959
6	1.1626	1.1382	1.1086	1.1534	1.1837

The data in Table 5.1 shows the ratio between the approximated upper limits to the spectrum versus the actual largest eigenvalue. The exact eigenvalues are computed by the *eigs* command in MATLAB. Essentially the ratios should be 1 or slightly over because an overestimation of the eigenvalues is better than an underestimation according to the Chebyshev iterative method theory.

Each row represents the different polynomial degrees, N. Each column represents the values of the different meshes. Five different meshes were used with

$$h = 0.2, 0.12, 0.1, 0.05, 0.025.$$

The results use the eigenvalues to this equation: $Su = \tilde{M}\lambda u$ where $\tilde{M}$ is the mass matrix. This work was done in MATLAB with code written by Warburton called $CurvedPoissonIPDG2D$ [2]. The meshes were generated from Jeomcad [41].

5.3 Approximated lower bound of the spectrum

This section briefly analyzes the lower bound to the variational problem. Recall the Lax-Milgram Theorem [1]:

Theorem 5.3.1. Lax-Milgram Theorem: *Let V_h be a real Hilbert space. Let $a_h : (V_h \times V_h) \to \mathbb{R}$ be a bilinear form that is*

- *continuous:* $|a_h(u,v)| \leq C_1 \|u\|_{DG} \|v\|_{DG}$
- *coercive :* $C_2 \|v\|_{DG}^2 \leq a_h(v,v)$

with positive constants C_1 and C_2.
Let $L : V_h \to \mathbb{R}$ be a continuous linear function. Then there exists a unique $u \in V_h$ satisfying

$$\forall v \in V_h \quad a_h(u,v) = L(v)$$

($\|\cdot\|_{DG}$ is defined below).

Define the $DG-norm$ as

$$\|v\|_{DG}^2 = \sum_{E\in T_h} \int_E K(\nabla v)^2 + \int_\Omega \alpha v^2 + \sum_{e\in\Gamma_h} \frac{\sigma_e}{|e|} \int_e [v]^2.$$

Assuming there is a unique solution to our problem, then by Theorem 5.3.1, a lower bound, based on the DG-norm, is given by the coercivity property. Rivière and Epshteyn [11], Shahbazi [15], and Ainsworth and Rankin [16], provide a result giving computable values for the constant C_2.

Also recall the next well-known theorem, [42]:

Theorem 5.3.2. Lagrange's Theorem: *Let V be a finite dimensional vector space over N, a normed vector space. Then any two norms on V are equivalent. Meaning that for $\|\cdot\|_a$, $\|\cdot\|_b$ two norms in this space, then there exists $D^*, D_* \in \mathbb{R}^+$ s.t.*

$$D_*\|z\|_a \leq \|z\|_b \leq D^*\|z\|_a \ \ \forall z \in V.$$

Applying the coercivity property and the finite dimensional vector space property, Theorem 5.3.2 implies that there exist constants, D_* and D^*, such that

$$D_*\|v\|_{L^2(\Omega)} \leq \|v\|_{DG} \leq D^*\|v\|_{L^2(\Omega)} \ \ \forall v \in V_h,$$

which also justifies that if the constant, D_*, is found, then by Theorem 5.3.1, the lower bound for the spectrum becomes

$$a_h(v,v) \geq C_2\|v\|_{DG}^2 \geq C_2 D_*^2 \|v\|_{L^2(\Omega)}^2.$$

The task is then to find the constant $C_2 D_*^2$. The proof of this bound remains an open problem. Analysis of the Poincaré-Friedrichs inequality can be found in [9], but

the constant not given explicitly. Also, an inverse bound to Theorem 5.2.1 can be formulated. The inverse bound for Theorem 5.2.3 has already been developed and can be found in [43]. More careful consideration is needed before a result can be given. This remains an open problem.

The next chapter will discuss the method for coding in CUDA.

Chapter 6

Method Used to Implement in CUDA

Now that analysis of the bounds of the DG spectrum applied to the elliptic problem are explored, the next task is to continue with the implementation of the linear system solvers in CUDA. Once these methods are developed, a previously written DG code provided by Rivière [1] and the iterative solvers are combined in order to find which method is more efficient when using CUDA.

6.1 Formatting implementation

The first step is to create code such that the DG matrix can be transformed in compressed sparse row (CSR) format. Rivière's code is written such that when forming the DG matrix, the matrix is stored in dense format and the linear system is solved using LAPACK routines. The code written for this thesis is converted to CSR format and is solved by the CG and Chebyshev iterative methods. This format is a popular, general-purpose sparse matrix representation. CSR stores its information in three vectors: *ptr*, *indices*, and *data*. For a $m \times n$ matrix, *ptr* is of length $m+1$ and stores the offset into the i^{th} row [32]. The *indices* vector stores the column number that the data is in. The *data* vector stores the values of the non-zero entries. Both the length of the indices vector and the data vector equal the number of nonzero entries in the matrix. An example from [32] is given below.

$$S = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \quad \begin{aligned} ptr &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ indices &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ data &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

Rivière's code is also in double precision. Recall from Chapter 3 that single precision provides significantly better peak performance than when using double precision. Therefore, to expect better performance from the CUDA framework, the data values are 'cast' down to single precision.

6.2 Implementing solvers using CUDA

Once the DG code has the correct sparse format, the next step is to implement the linear solvers using CUDA. This is slightly different than the standard C code, because different commands are needed to help communicate with the GPU.

To inform the GPU to execute commands, there is a function called a kernel. The kernel is executed using a large number of parallel threads, where each thread runs the same commands simultaneously. The GPU and CPU have separate memory pools, i.e., information on the GPU cannot be accessed from the CPU and vice-versa. This implies that when using the GPU, space must be allocated before information from the CPU can be sent to the GPU; refer to Figure 6.1. Once the data is on the GPU, the kernel is called to execute the commands. Finally the GPU sends the information back to the CPU. The programmer must make sure to perform few transfers from the CPU to the GPU in order to reduce the bottleneck caused by the transfer of the information. For the iterative method implementations, CUBLAS commands were used. As an example of the protocol, say there exists a vector *h_mat*, of length *nnz*

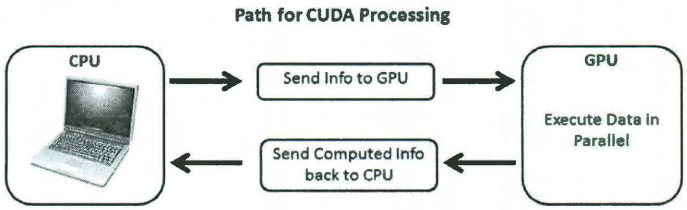


Figure 6.1 : Procedure when communicating with the GPU

on the host (CPU). To send this vector to the device (GPU), another vector, *d_mat*, must be first be allocated, then information from *h_mat* can be transferred to *d_mat*. The commands to allocate space on the device and send the information from the host to the device are seen below [3, 33].

```
cublasAlloc(nnz, sizeof(float), (void**) &d_mat)
cublasSetVector(nnz, sizeof(float), h_mat, 1, d_mat, 1)
```

A kernel is then called to perform the necessary operations on the GPU. To review the result found by the GPU, the information is sent back to the CPU with the CUBLAS command,

```
cublasGetVector(nnz, sizeof(float), d_mat, 1, h_mat, 1).
```

Once the CPU has the information, commands can continue to be executed on the CPU or results can be printed. Next step is understanding how the kernel is executed.

6.3 Structure of GPU

Define a thread as being the smallest unit that can execute a command; a a warp is a collection of 32 threads. Within this language, these threads can be organized using grids of threads. A thread block is defined as a group of threads that share a common

pool of shared memory and cooperate together efficiently [44]. Each thread block has a maximum of 512 threads. Blocks of the same dimension that execute the same kernel commands are grouped together in a grid block. Threads of different blocks share a different memory pool. In other words, the threads in each block cannot communicate and synchronize with each other.

All of these threads and blocks have thread identifications. CUDA has built in variables that help to describe the identification for each of the threads. For example a 1-D block and grid, has identifications like,

```
blockDim.x  \\the dimension in the grid
blockIdx.x  \\the ID of the block within in the grid
threadIdx.x  \\the ID of the thread within the block
```

To access the complete identification of a particular thread, the command is

```
int thread_id = blockDim.x*blockIdx.x + threadIdx.x;
```

These threads can be formulated in 1-D, 2-D, or 3-D structure. For example, to inform the kernel to use a 1-D block of size nnz with nnz threads in each block, before the kernel is called [3], type the commands,

```
dim3 dimThreadBlock(nnz) \\Dimension of the block
dim3 dimGridBlock(nnz);  \\Dimension of the grid
```

To call the kernel function, $MV\_multiply$, the dimensions of the blocks and grids must be given by using $<<<, >>>$.

```
MV_multiply<<< dimGridBlock, dimThreadBlock >>>(d_ptr, d_indices);
```

The remainder of the variables, $(d\_ptr, d\_indices)$, are input and output arguments needed for that kernel. These variables are already allocated on the GPU.

The algorithms in Appendix A are used when implementing the kernels for the two linear system solvers. In order to perform inner products, the CUBLAS command, *cublasSdot*, is called. Bell and Garland's implementation of CSR matrix vector multiplication is used in both CG and the Chebyshev iterative codes [32]. Kernels needed to compute the rest of the calculations are created. The next chapter provides the results for when these iterative solvers were applied to Rivière's DG code for solving the Laplace problem.

Chapter 7

Numerical Results

This chapter provides numerical results that incorporate the implementation described in Chapter 6 and an existing DG code, written in C, provided by Rivière [1]. These numerical results determine which iterative solver is best suited when using CUDA.

The code given by Rivière [1] solves the Laplace equation,

$$-\Delta u = f$$

with Dirichlet boundary conditions on a square domain $[0,1] \times [0,1]$. The Chebyshev iterative method and the CG method written in the languages CUDA and C, are applied to Rivière's DG code.

Table 7.1 : Meshes used throughout the numerical testing.

Meshes	Num of Elem	h	Dof $N = 1$	Dof $N = 2$
mesh1	4	1	12	24
mesh2	22	0.2500	66	132
mesh3	110	0.1250	330	660
mesh4	544	0.0625	1632	3264
mesh5	1168	0.0450	3504	7008
mesh6	2190	0.03125	6570	13140
mesh7	3630	0.0250	10890	21780
mesh8	5440	0.0200	16320	32640

For these simulations, the Tesla C2050 card along with the CPU by AMD Opteron(tm) Processor 148 is used. Eight different triangular meshes are generated. These meshes are created from an element grid generator called Gmsh [45]. Table 7.1 describes the range of h, the number of elements, and the degrees of freedom for each mesh when performing the DG method. Images of the meshes used can be found in Figure 7.1.

Table 7.2 : Eigenvalues used corresponding to different σ_e, $N = 1$, and meshes.

Meshes	max eig $\sigma_e = 10$	min eig $\sigma_e = 10$	max eig $\sigma_e = 100$	min eig $\sigma_e = 100$	max eig $\sigma_e = 1000$	min eig $\sigma_e = 1000$
mesh1	20	0.9339	200	0.9949	2000	0.9995
mesh2	19.41	0.2803	194.68	0.2906	1945.5	0.2918
mesh3	19.67	0.0636	196.997	0.0639	1970.3	0.0639
mesh4	19.78	0.0110	198.02	0.0110	1980.4	0.0110
mesh5	19.89	0.0056	198.98	0.0056	1989.9	0.0056
mesh6	19.80	0.0029	198.23	0.0029	1982.5	0.0029
mesh7	19.96	0.0017	199.67	0.0017	1996.8	0.0017
mesh8	19.95	0.0010	199.54	0.0010	1995.5	0.0010

Before describing the results, first note that the tolerance was set at 10^{-4} for both methods. When solving the Chebyshev iterative method, the eigenvalues were found by MATLAB's *eigs* command. For the larger problems where MATLAB could not load the matrices, approximated values were found using LAPACK'S *dgeev* command. Tables 7.2 and 7.3 present the eigenvalues for this study.

Additionally, two different parameters were varied, σ_e and N. Tests were run for all eight meshes, $\sigma_e = 10, 100, 1000$, and $N = 1, 2$. Similar results are found through all the runs, therefore, the first six images correspond for when $\sigma_e = 100$ and $N = 1$.

Figure 7.2 gives the times it takes for the CG method and the Chebyshev iterative

Table 7.3 : Eigenvalues used corresponding to different σ_e, $N = 2$, and meshes.

Meshes	max eig $\sigma_e = 10$	min eig $\sigma_e = 10$	max eig $\sigma_e = 100$	min eig $\sigma_e = 100$	max eig $\sigma_e = 1000$	min eig $\sigma_e = 1000$
mesh1	25.91	0.0182	270.95	0.0439	2722.2	0.0475
mesh2	24.76	0.0126	258.85	0.0169	2600.9	0.0173
mesh3	25.40	0.0120	263.73	0.0149	2647.6	0.0151
mesh4	25.57	0.0095	265.45	0.0098	2664.7	0.0098
mesh5	25.53	0.0053	264.92	0.0054	2660.3	0.0054
mesh6	25.63	0.0031	267.35	0.0031	2665.3	0.0030
mesh7	25.66	0.0020	266.59	0.0022	2666.2	0.0022
mesh8	25.70	0.0012	268.43	0.0011	2668.4	0.0012

method to solve the DG linear system. All values calculate only the time it takes for the linear system to solve. Time was not considered when finding maximum and minimum eigenvalues and also when formulating the DG matrix. It can easily be seen from Figure 7.2 that the Chebyshev iterative method takes at least two orders of magnitude longer to compute compared to the CG method. This implies that the CG method outperforms the Chebyshev iterative method. It seems counterintuitive that the Chebyshev iterative method takes longer to converge when it does not have any inner products. The values below show that time it takes for the CG method and the Chebyshev iterative method to complete one iteration. Clearly, because of the inner products, the CG method takes longer than the Chebyshev iterative method. Notice that this is for a $16,320 \times 16,320$ matrix, $N = 1$, and $\sigma_e = 100$.

```
the size of the matrix is 16320
elapsed time for CG: 0.002110
elapsed time for the Chebyshev iterative method: 0.000760
```

```
iteration count = 1
```

The next set of values provides a breakup of time between the different kernels for the CG method. These values are accumulations for each iteration. Again this is a $16,320 \times 16,320$ matrix, and the two inner products take longer to compute than the matrix vector multiplication kernel.

```
the size of the matrix is 16320
number of non zeros = 185168
elapsed time for Matrix Vector Multiply kernel: 0.267620
elapsed time for cublasSdot kernel: 0.194517
elapsed time for Update1 kernel: 0.053182
elapsed time for cublasSdot kernel: 0.192549
elapsed time for Update2 kernel: 0.050227
elapsed time for Total time: 1.175548
iteration count = 689
gflops = 0.312711
```

The reason why the CG method outperforms the Chebyshev iterative method is due to the large number of iterations it takes for the Chebyshev iterative method to obtain the desired accuracy. Figure 7.3 expresses that the number of iterations for the Chebyshev iterative method is drastically higher compared to the CG method. For example, for the last mesh, mesh8, with matrix dimension size $16,320$, it takes about $546,739$ iterations for the Chebyshev iterative method to converge while for the CG method, only 689 iterations are needed. This is 3 orders of magnitude difference.

The next two graphs give more information on the convergence of the two methods and the eigenvalue distribution. Figure 7.4 and Figure 7.5 give only information for

when $\sigma_e = 100$, $N = 1$, and mesh5. Figure 7.4 compares the residuals for each iteration. By looking at this graph, the conjugate gradient method is clearly converging at a much faster rate every iteration. For the Chebyshev iterative method, there is an initial jump down, then there is a slow decrease in the residual deceasing at an average rate a value slightly below 1, i.e. 0.9967. The Chebyshev iterative method took 100243 iterations to converge versus the conjugate gradient method taking 547 iterations. In the figure, only residuals for a thousand iterations were portayed. Figure 7.5 provides the distribution for the spectrum. In this graph, you can see where the clusters of eigenvalues are. The CG method converges faster when there are clusters of eigenvalues. This explains the faster convergence rate in Figure 7.4.

Also, the different hardware is compared in the next two figures. Times for the CG implementations in CUDA and C are given in Figure 7.6 for the eight different meshes. Notice, for the first three meshes, the C code is faster than the CUDA code. This is due to the lag time it takes for information to transfer back and forth between the CPU and GPU. For mesh8, there was at least an order of magnitude difference in the computed time. This shows that using the GPU for large problems decreases the time drastically compared to the conventional CPU. Similar results are found in Figure 7.7 comparing the times for the Chebyshev iterative method to converge for the two languages. It was not until the fourth mesh, where the matrix is relatively big, when work on the GPU is completed faster than the CPU.

Figure 7.8 gives a graph conveying the Gflops/sec for each different mesh and each solver executed using CUDA. The performance increases linearly as the number of elements increase and the degrees of freedom increase. In viewing these numbers, the performance is very small. To figure out why the codes doh not seem very efficient, a different algorithm is compared. CUSP is a library for sparse linear algebra and graph

computations on CUDA and, within this library, the CG algorithm is implemented [46]. Figure 7.9 shows the Gflops/sec comparing the CG method using my algorithm and the CUSP algorithm. The values are very similar. I suspect that if using larger matrices, $100,000 \times 100,000$, the performance will be more promising.

The last two figures compare the penalty parameters for a specific polynomial degree and a given mesh. Time values from mesh5 are given in Figure 7.10 and 7.11. Again, results from the CG method and the Chebyshev iterative method solved on the GPU are given. Figure 7.10 shows that as the penalty parameter increases, the times it takes for both methods to converge also increases. The same is true for changing the polynomial degree, as seen in Figure 7.11. Comparing Figures 7.10 and 7.11 side by side, notice that as the polynomial degree gets bigger, the time will also increase for both methods to converge. Tests for higher degree polynomials are currently unavailable until alterations in the DG C code can be made to formulate the stiffness matrix into sparse format versus dense. Similar results are given for the different meshes; therefore those results are not included.

Various tests were given in this results section. Comparisons between the Chebyshev iterative method and the CG method using CUDA are given. Tests also showed the difference of the two methods using C and CUDA for both methods. To make sure the code was working efficiently, the CG CUDA code is compared to the CUSP library, which also solves the CG method in CUDA. Last, two different parameters were varied to see how computing the methods were affected. From these results, CG is the more effective solver for the GPU compared to the Chebyshev iterative method for larger problems. This is because the Chebyshev iterative method uses many more iterations compared to the CG method, i.e. enough iterations to increase the time to surpass what it takes to complete the inner products needed in the CG method. I

also showed, from Figures 7.6 and 7.7 that with large data sets, the GPU speeds up computational time. Ideally, these tests were to be conducted with approximations to the maximum and minimum eigenvalues applied to the Chebyshev iterative method. However, after running numerical simulations with exact eigenvalues, there is no need to continue the study. The next chapter will provide some concluding remarks.

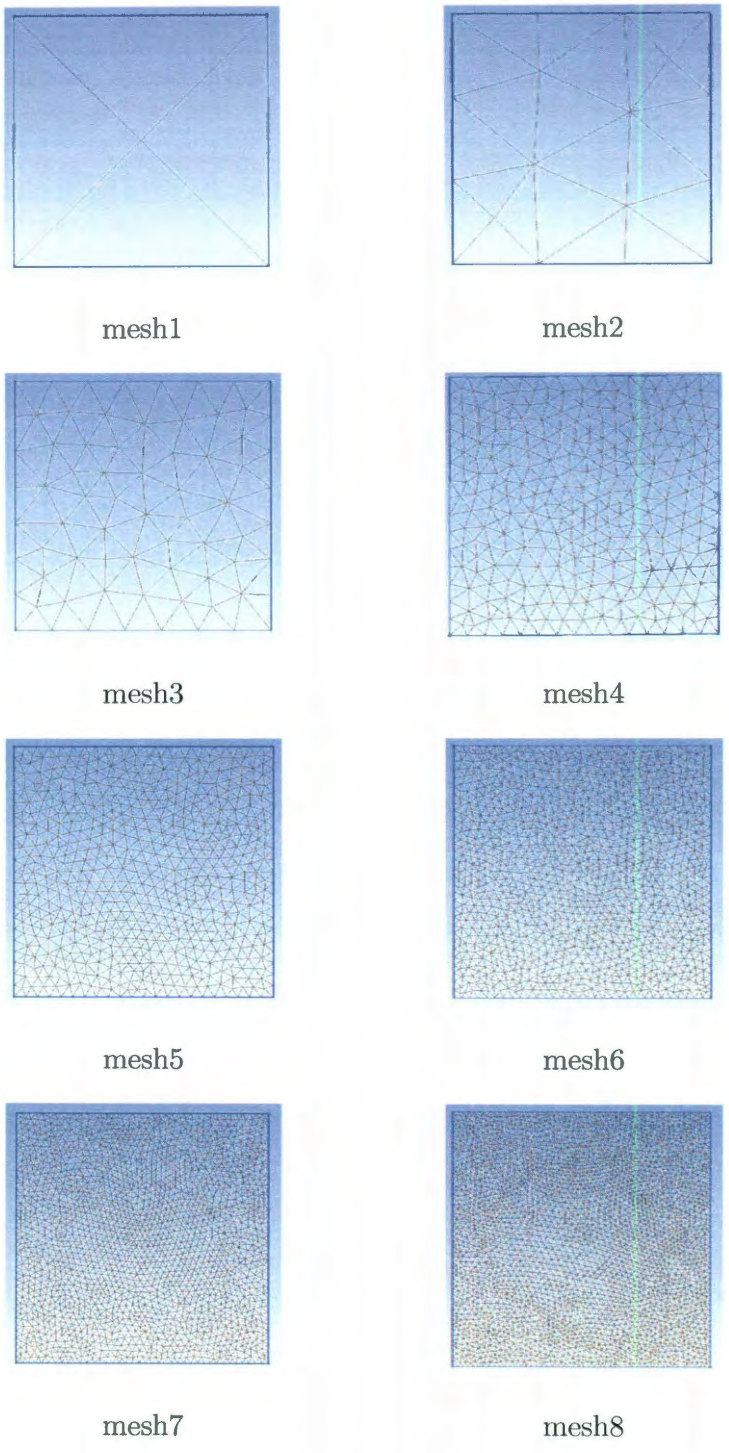


Figure 7.1 : Meshes created by Gmsh used in the experiment

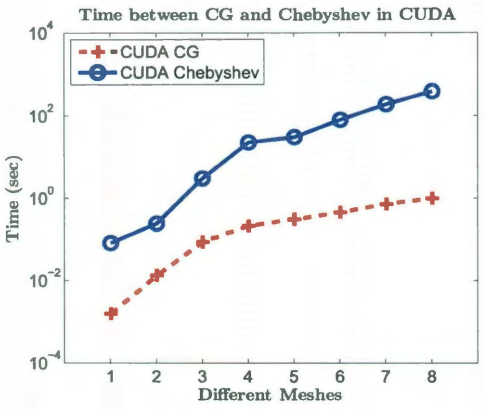


Figure 7.2 : Time comparing the CG vs. the Chebyshev iterative method using CUDA where $\sigma_e = 100$ and $N = 1$

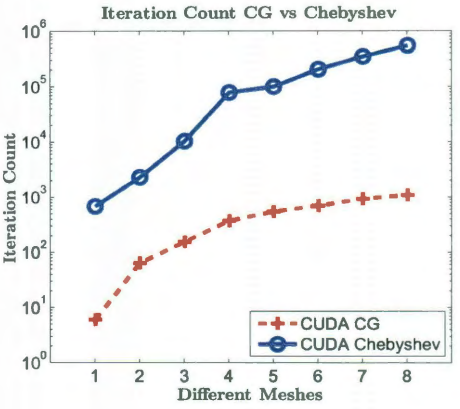


Figure 7.3 : Iteration count comparing the CG vs. the Chebyshev iterative method using CUDA where $\sigma_e = 100$ and $N = 1$

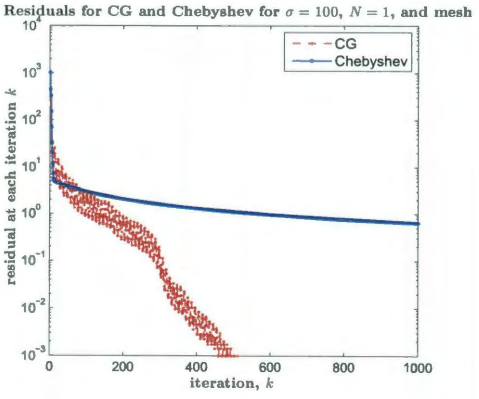


Figure 7.4 : Residuals for CG and the Chebyshev Iterative method at every iteration for $\sigma_e = 100$, $N = 1$, and mesh5

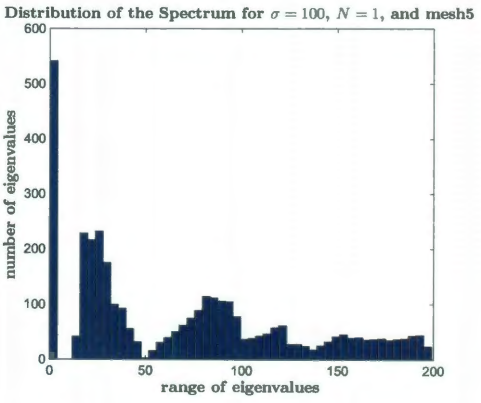


Figure 7.5 : Distribution of the eigenvalues for $\sigma_e = 100$, $N = 1$, and mesh5

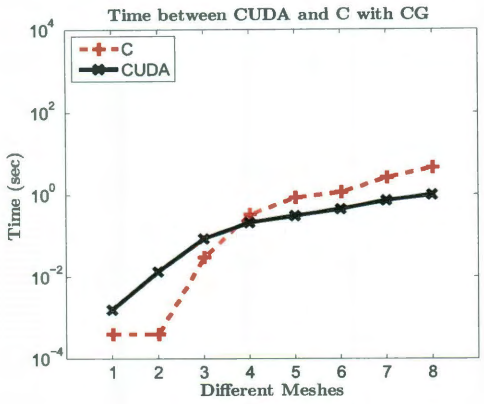


Figure 7.6 : Times in CUDA and C for CG method to converge where $\sigma_e = 100$ and $N = 1$

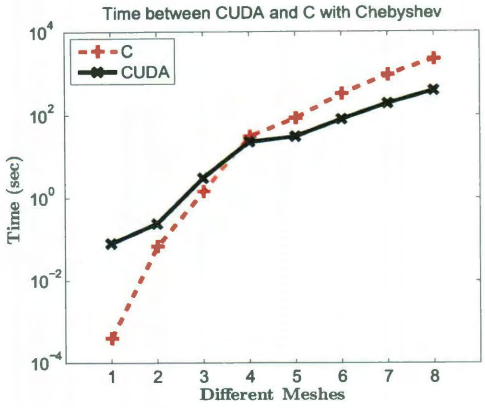


Figure 7.7 : Times in CUDA and C for Chebyshev iterative method to converge where $\sigma_e = 100$ and $N = 1$

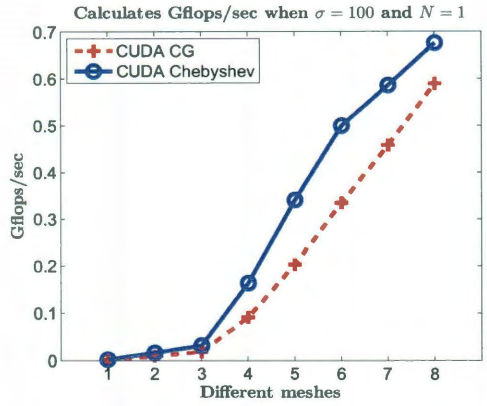


Figure 7.8 : The Gflops/sec calculated for both CG and the Chebyshev iterative method executed using CUDA with parameters were set to $\sigma_e = 100$ and $N = 1$

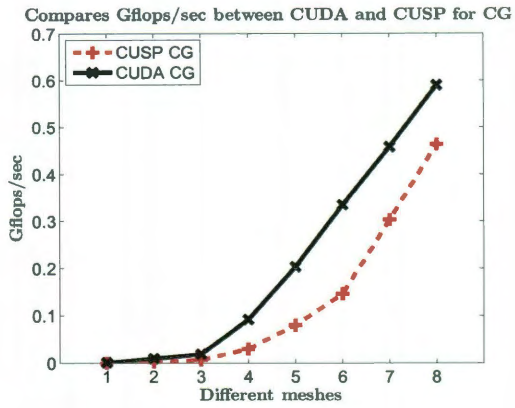


Figure 7.9 : The Gflops/sec calculated for CG executed using CUDA and CUSP with parameters were set to $\sigma_e = 100$ and $N = 1$

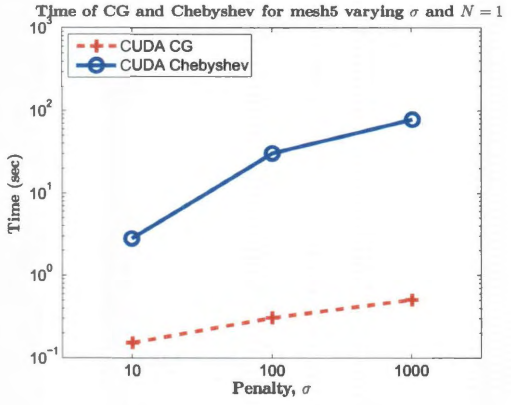


Figure 7.10 : Comparing time for both methods while varying σ_e when $N = 1$ using CUDA

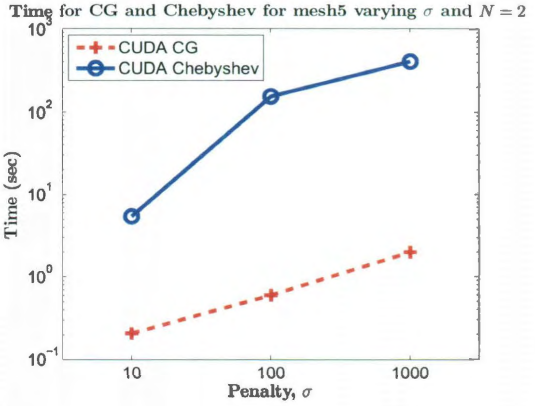


Figure 7.11 : Comparing time for both methods while varying σ_e when $N = 2$ using CUDA

Chapter 8

Conclusions

This work integrates the use of supercomputers, the DG method, and an eigenvalue problem. There are two main ideas presented in this research. The first was to analyze approximations for the maximum and minimum eigenvalues of the DG operator applied to the elliptic equation. A constant-free bound for the maximum eigenvalue is developed. The minimum eigenvalue bound remains an open problem.

The second goal was to implement the Chebyshev iterative method and the CG method using C and CUDA, a software library that communicates with the GPU. A comparison was made between the two methods combined with the DG method when applied to the elliptic problem in order to determine which proved to be the most effiecint method using the GPU framework. The CG method, a linear system solver with two inner products per iteration, is more effective for larger problems versus the Chebyshev iterative method, an algorithm with no inner products. This is due to the fact that the Chebyshev iterative method takes many more iterations to converge, which increased the time needed to compute so much that the time it takes to complete the inner products of the CG method was shorter.

There are two directions to proceed with this work. First, complete the analysis for the lower eigenvalue of the variational problem. Another direction for this work is to test two other methods similar to both the CG method and the Chebyshev iterative method. The first method is to use these Chebyshev polynomials up to degree r, then perform the CG method until the residual converges to a given tolerance. The reason

for performing this pre-conditioned method is because in parallel computing, the Chebyshev iterative method can perform for a preset number of times, without any use of an inner product, then use the CG method to converge to the solution in fewer iterations, implying fewer inner products. The other method is to first estimate the eigenvalues using the Lanczos/CG method for the first k iterations. Starting with the $k+1$ iteration, the Chebyshev iterative method is used to converge to the solution. By using the CG method first, initializing the eigenvalues is not necessary. Therefore this method avoids having over/under estimates for the eigenvalues which may or may not allow the Chebyshev iterative method to converge. Also, this algorithm reduces the times the CG method is performed compared to the standard CG method; therefore the number of inner products are again reduced.

One area of study, which is of particular interest, that can benefit from this thesis is the area of heat transfer, solving the steady state conduction equation. With this work, engineers with heat transfer specialties can be more efficient in solving for temperature profiles through a heat sink when a heat flux is applied. This is only one of many applications in which this work will contribute.

This thesis is one of the few works that combine the ideas of the DG method and CUDA. It provides mathematicians and engineers knowledge of which numerical linear system solver is more effective between the two tested when applying DG to the elliptic problem. It also provides the proof for a constant free upper bound of the DG spectrum as it is applied to the elliptic problem.

Bibliography

[1] B. Rivière, *Galerkin Methods for Solving Elliptic and Parabolic Equations: Theory and Implementation.* Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2008.

[2] J. S. Hesthaven and T. Warburton, *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications.* New York, New York: Springer Science+Business Media, LLC, 2008.

[3] "NVIDIA CUDA, programming guide," *NVIDIA Corporation*, vol. Version 3.0, Feburary, 2010. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf.

[4] A. Cevahir, A. Nukada, and S. Matsuoka, "Fast conjugate gradients with multiple GPUs," in *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, (Berlin, Heidelberg), pp. 893–903, Springer-Verlag, 2009.

[5] C. C. Douglas, G. Haase, and U. Langer, *A tutorial on elliptic PDE solvers and their parallelization.* Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2003.

[6] S. Brenner and L. R. Scott, *The Mathematical Theory of Finite Element Methods.* New York, New York: Springer Science+Business Media, LLC, 2008.

[7] H. K. Versteeg and W. Malalasekera, *An Introduction to Computational Fluid Dynamics, The finite Volume Method.* London: Prentice Hall, 1995.

[8] W.H. Reed and T. Hill, "Triangular Mesh Methods for the Neutron Transport Equation," *Technical Report LA-UR-73-479, Los Alamos Scientific Laboratory*, 1973.

[9] D. N. Arnold, "An interior penalty finite method with disontinuous elements," *SIAM Journal on Scientific and Statistical Computing*, vol. 19, no. 4, pp. 742–760, 1982.

[10] D. Arnold, F. Brezzi, and B. Cockburn, "Discontinuous Galerkin methods for elliptic problems," in *Discontinuous Galerkin Methods (Newport, RI, 1999), Lecture Notes Computational Science Engineering*, p. 89, Springer, 2000.

[11] Y. Epshteyn and B. Rivière, "Estimation of penalty parameters for symmetric interior penalty Galerkin methods," *J. Comput. Appl. Math.*, vol. 206, no. 2, pp. 843–872, 2007.

[12] D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini, "Unified analysis of discontinuous Galerkin methods for elliptic problems," *SIAM J. Numer. Anal.*, vol. 49, no. 5, pp. 1749–1779, 2002.

[13] B. Cockburn, G. E. Karniadakis, and C. W. Shu, "The development of discontinuous Galerkin methods," 1999. http://conservancy.umn.edu/handle/3384.

[14] P. F. Antonietti, A. Buffa, and I. Perugia, "Discontinuous Galerkin approximation of the Laplace eigenproblem," *Computer Methods in Applied Mechanics and Engineering*, vol. 195, no. 25-28, pp. 3483 – 3503, 2006.

[15] K. Shahbazi, "Short note: An explicit expression for the penalty parameter of the interior penalty method," *J. Comput. Phys.*, vol. 205, pp. 401–407, May 2005.

[16] M. Ainsworth and R. Rankin, "A note on the selection of the penalty parameter for discontinuous Galerkin finite element schemes," *Numerical Methods for Partial Differential Equations*, vol. 27, 2011.

[17] T. Warburton and J. S. Hesthaven, "On the constants in hp-finite element trace inverse inequalities," *Computer Methods in Applied Mechanics and Engineering*, vol. 192, no. 25, pp. 2765 – 2773, 2003.

[18] S. Ozisik, B. Rivière, and T. Warburton, "On the constants in inverse inequalities in l^2," tech. rep., Rice University, 2010.

[19] "PTIGlobal chosen sole localization vendor to NVIDIA," 2009. http://www.ptiglobal.com/whats_new/story/ptiglobal_chosen_sole_localization_vendor_to_nvidia_14.

[20] "Graphics processing unit," 2011. http://en.wikipedia.org/wiki/Graphics_processing_unit.

[21] "NVIDIA GeForce 256 review:GPU overview," *NVIDIA Corporation*, 1999. http://www.nvnews.net/reviews/geforce_256/gpu_overview.shtml.

[22] "Tesla c2050/c2070 GPU computing processor," *NVIDIA Corporation*, 2011. http://www.nvidia.com/object/product_tesla_C2050_C2070_us.html.

[23] W. A. Wiggers, V. Bakker, A. B. J. Kokkeler, and G. J. M. Smit, "Implementing the conjugate gradient algorithm on multi-core systems," in *Proceedings of the*

International Symposium on System-on-Chip (SoC 2007), Tampere (J. Nurmi, J. Takala, and O. Vainio, eds.), no. 07ex1846, (Piscataway, NJ), pp. 11–14, IEEE, November 2007.

[24] "What is GPU computing," *NVIDIA Corporation*, 2011. http://www.nvidia.com/object/GPU_Computing.html.

[25] Y. Zhao, "Lattice Boltzmann based PDE solver on the GPU," *Vis. Comput.*, vol. 24, pp. 323–333, May 2008.

[26] D. Egloff, "High Performance Finite Difference PDE Solvers on GPUs," February 2010. http://gpucomputing.net/?q=node/1380.

[27] C. Becker, S. Turek, and S. Kilian, "Feast: Finite Element Analysis and Solutions Tools," http://www.feast.tu-dortmund.de/index.html.

[28] A. Kloeckner, T. Warburton, J. Bridge, and J. Hesthaven, "Nodal discontinuous galerkin methods on graphics processors," 2011. http://mathema.tician.de/software/pycuda.

[29] H. Dag, "An approximate inverse preconditioner and its implementation for conjugate gradient method," *Parallel Computing*, vol. 33, no. 2, pp. 83 – 91, 2007.

[30] H. Dag and A. Semlyen, "A new preconditioned conjugate gradient power flow," *Power Systems, IEEE Transactions on*, vol. 18, pp. 1248 – 1255, nov. 2003.

[31] R. Li and Y. Saad, "Gpu-accelerated preconditioned iterative linear solvers *." http://www-users.cs.umn.edu/ saad/PDF/umsi-2010-112.pdf.

[32] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," Tech. Rep. NVR-2008-004, NVIDIA Technical Report, Dec. 2008.

[33] "CUDA, CUBLAS library," *NVIDIA Corporation.*

[34] D. P. O'Leary, "Yet another polynomial preconditioner for the conjugate gradient algorithm," *Linear Algebra and its Applications*, vol. 154-156, pp. 377 – 388, 1991.

[35] L. N. Trefethen and D. Bau III, *Numerical Linear Algebra.* Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 1997.

[36] Y. Saad, *Iterative Methods for Sparse Linear Systems, Second Edition.* Society for Industrial and Applied Mathematics, 2003.

[37] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd ed.).* Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[38] G. H. G. Concus, Paul and D. P. O'Leary, "A generalized conjugate gradient method for the numerical solution of elliptic partical differential equations," Tech. Rep. STAN-CS-76-533, Stanford : Computer Science Dept., School of Humanities and Sciences, Stanford University, 1976.

[39] A. Wathen and T. Rees, "Chebyshev semi-iteration in preconditioning," *Electronic Transactions on Numerical Analysis*, vol. 34, pp. 125–135, 2009.

[40] P. Concus, G. H. Golub, and D. P. O"Leary, "A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations," tech. rep., Stanford, CA, USA, 1976.

[41] T. Issac, L. Wilcox, and T. Warburton, "Jeomcad," http://www.caam.rice.edu/ timwar/MeshGeneration/Jeomcad.html.

[42] S. Lang, *Undergraduate Analysis.* New York, New York: Springer Science+Business Media, LLC, 1983.

[43] E. Burman and A. Ern, "Continuous interior penalty hp-finite element methods for advection and advection-diffusion equations," 2010. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.153.7434.

[44] N. Fujimoto, "Faster matrix-vector multiplication on geforce 8800gtx," *IEEE International Symposium on In Parallel and Distributed Processing*, pp. 1–8, 2008.

[45] C. Geuzaine and J. F. Remacle, "Gmsh," 1997-2009. http://geuz.org/gmsh/.

[46] N. Bell, "CUSP," 2011. http://code.google.com/p/cusp-library/.

[47] Y. Saad, "Krylov subspace methods on supercomputers," *SIAM Journal on Scientific and Statistical Computing*, vol. 10, no. 6, pp. 1200–1232, 1989.

Appendix A

A.1 Algorithms

In this Appendix the codes for the two numerical solvers are provided: CG and the Chebyshev iterative method.

A.1.1 Code for the CG method:

More information about this method can be found from Saad [47].

```
% This method computes Ax = b using the conjugate gradient method
% input     A     a MxM symmetric Positive Definite Matrix
%           b     a Mx1 vector
%           x     a Mx1 vector, initial guess
%           tol   the stopping critiera
% Output    x     the solution
%           r     the residual
%           i     the number of iterations performed
r = b - A*x;
p = r;
rold = r'*r;
for i = 1:size(A,1)
    Ap = A*p;
    alpha = rold/(p'*Ap);
    x = x + alpha*p;
    r = r - alpha*Ap;
    rnew = r'*r;
```

```
18      if(norm(r))< tol
19          break;
20      end
21      p = r + (rnew/rold)*p;
22      rold = rnew;
23  end
```

A.1.2 Code for the Chebyshev iterative method

This method can be found in many locations. I referenced Saad [36] and Golub [37].

```
1  function [x, r, i]= cheby_iter(A,x,b,L_max,L_min,maxit,tol)
2  % this function solves Ax = b for x in R^n
3  % it is an iterative method that does not use an inner product
4  % but needs the L_max and L_min
5
6  % input     A       A is an nxn matrix
7  %           x       x is the initial guess
8  %           b       right hand side nx1
9  %           M       preconditioner matrix nxn
10 %                   do not have this in because just identity for now
11 %           L_max   max eigenvalue of inv(M)*A
12 %           L_min   min eigenvalue of inv(M)*A
13 %           maxit   max number of iterations
14 %           tol     tolerance for convergence
15 % output    x       final solution
16 %           r       residual
17 %           i       number of iterations used
18 r = b - A*x;
19 d = (L_max + L_min)/2;
20 c = (L_max - L_min)/2;
```

```
21 for i = 1:maxit
22     z = r;
23     if i == 1
24         p = z;
25         alpha = 2/d;
26     else
27         beta = alpha*alpha*c*c/4;
28         alpha = 1/(d-beta);
29         p = z + beta*p;
30     end %end if statement
31     x = x + alpha*p;      %perform linesearch
32     r = b - A*x; %r = r - alpha*A*p;
33     if norm(r) < tol
34         break
35     end      %end of if
36 end %end of for
```

Appendix B

B.1 Supplementary proofs

In this Appendix proofs for some parameters which are useful in helping to prove the maximum bound to the variational problem are given. These values are dependent of the mesh used. These supplementary proofs were referenced in Chapter 5.

B.2 Proof for constants $\frac{|\partial E|}{|E|}$ and $\frac{|e|}{|E|}$

Observe the picture of a triangular element. This image will be useful in proving the values.

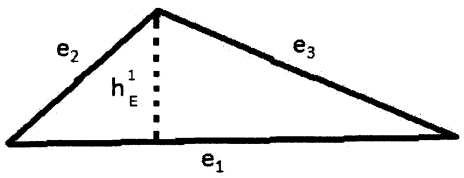


Figure B.1 : An arbitrary triangle element, E, in the mesh

B.2.1 Value for $\frac{|e|}{|E|}$

Let e be the edge corresponding to the i^{th} side in element; E. Also observe that $|E| = \frac{1}{2} e_i h_E^i$, refer to Figure B.1. Then

$$\frac{|e|}{|E|} = \frac{e_i}{\frac{1}{2} e_i h_E^i} = \frac{2}{h_E^i}.$$

B.2.2 Bound for $\frac{|\partial E|}{|E|}$

Based on Figure B.1, the perimeter, $|\partial E|$, and area, $|E|$, of the element can be defined as

$$\begin{aligned} |\partial E| &= e_1 + e_2 + e_3, \\ |E| &= \frac{1}{2} e_1 h_E^1 = \frac{1}{2} e_2 h_E^2 = \frac{1}{2} e_3 h_E^3, \end{aligned}$$

respectively. This implies

$$\begin{aligned} \frac{|\partial E|}{|E|} &= \frac{e_1 + e_2 + e_3}{|E|} \\ &= \frac{e_1}{|E|} + \frac{e_2}{|E|} + \frac{e_3}{|E|} \\ &= \frac{e_1}{\frac{1}{2} e_1 h_E^1} + \frac{e_2}{\frac{1}{2} e_2 h_E^2} + \frac{e_3}{\frac{1}{2} e_3 h_E^3} \\ &= \frac{2}{h_E^1} + \frac{2}{h_E^2} + \frac{2}{h_E^3} \\ &= 2 \sum_{i=1,2,3} \frac{1}{h_E^i}. \end{aligned}$$